EDTASM+ ™

COLOR COMPUTER
EDITOR ASSEMBLER
WITH ZBUG

TABLE OF CONTENTS

1 / Introduction

The brain of the Color Computer is the 6809 Microprocessor. It is always operating in 6809 machine code, the only language it knows.

When you program in BASIC, a ROM program called the BASIC Interpreter "translates" each statement, one at a time, into 6809 machine code.

The Editor-Assembler ÷ allows you to write a program in 6809 assembly language and assemble it into a single, efficient 6809 machine code program. This gives you two very powerful advantages:

• You are no longer limited to the commands in the BASIC language.

• Many steps that are necessary to interpret a BASIC statement into machine code will no longer be needed. Therefore, the programs you write with the Editor-Assembler+ will run much faster, and probably use less memory.

This manual demonstrates how to use the Editor-Assembler+. It will not teach you how to program in assembly language. Radio Shack has an excellent book devoted to the subject. It's Catalog Number is 62-2077. You can purchase it through any Radio Shack store.

The Editor-Assembler ÷ contains three systems:

• *The Editor*, for writing and editing 6809 assembly language programs.

• *The Assembler*, for assembling the programs into 6809 machine code.

• *ZBUG*, for examining and debugging your machine code programs.

To use them, all you need is a Color Computer with 16K RAM and a tape recorder.

How You Will Use These Systems

1. First you'll *write the program* in assembly language, using mnemonics which the Assembler recognizes and which is fairly easy to use. This is done in the Editor and the resulting program listing is called TEXT.

2. Then you'll *assemble* the instructions of TEXT into machine code which the 6809 Microprocessor can recognize, but which looks like nonsense to most people. Thus, you'll create CODE consisting of op codes and data.

3. You'll use ZBUG to *test and debug* CODE until it's perfect. Then you'll store it on tape. Storing CODE is the final task of the Editor-Assembler ÷.

4. From BASIC, you'll *load* CODE (with CLOADM) *and run it.* You can either run it as a stand-alone program (with EXEC) or as a subroutine (with USR).

How This Manual Will Guide You

This manual will walk you through all these steps and also give you some useful information about your Editor-Assembler+.

In *Chapter 2*, we'll explore memory. You'll need this foundation to understand the rest of the manual. We'll do this with ZBUG.

Chapters 3, *4*, *5*, and *6* will show you how to write the program, assemble it, and debug it. Finally in *Chapter 7*, we'll show you how to run the program from BASIC.

If you've used other editor-assemblers, you might want to start with the *Appendixes*. There, you'll find all the commands summarized with page number references.

And Now Let's Get On With It . . .

To use the Editor-Assembler+, follow these steps:

1. Turn OFF the computer.

2. Insert the ROM pack into the slot on the right side of the computer.

3. Turn the Computer ON.

When you turn the computer ON, you will see:

```
EDTASM+ 1.0
COPYRIGHT © 1981 BY MICROSOFT

*
```

The asterisk prompt (*) tells you that the Editor is now available. We say you are "in" the Editor.

2 / Examining Memory

To use the Editor-Assembler +, you must have a good understanding of the Color Computer's memory. You will need to know about memory to write the program, assemble it, debug it, and execute it.

In this Chapter, we'll explore memory and see some of the many ways you can get the information you want. To do this, we'll use ZBUG.

Type:

Z (ENTER)

and ZBUG will display its # prompt. You are now "in" ZBUG and you may enter a ZBUG command.

All ZBUG commands must be entered in this command level. You can return to it by pressing (BREAK) or (ENTER).

Examining a Memory Location

The 6809 can address 65,536 one-byte memory locations, numbered 0-65535 (0000-FFFF hexadecimal). We'll examine hexadecimal location C000, the beginning of the Editor-Assembler program. Type:

C000/

LDA #6 is the "mnemonic" instruction that begins at location C000.

To examine the next instructions, press the (↓). Use the (↑) to get back to a preceding location. Notice that when you use the (↑) the screen continues to scroll down. The smaller addresses are displayed at the bottom of the screen.

Also notice that the (↓) will increment by more than one byte in this particular examination mode. More on this in the following pages.

The (↑), however, will always decrement the address by one, regardless of the examination mode.

All the numbers you see are hexadecimal. Hexadecimal means base 16. You will see not only the ten numeric digits, but also the six alpha characters needed for base 16 (A-F). ZBUG assumes you want to see base 16 numbers unless you specify another base (which we'll do in Chapter 6).

Notice that a zero precedes all the hexadecimal numbers beginning with an alphabetic character. This is done to avoid any confusion between hexadecimal numbers and registers.

Examining Modes

To help you interpret the contents of memory, ZBUG offers four ways of looking at it:

Byte Mode

Type (BREAK) to get into the command level and then type:

B (ENTER)

Examine the contents beginning at location C000 again. LDA #6 is now represented as a number. 86 is the op code for LDA. The operand, 6, is in location C001.

The byte mode displays every byte of memory as a number, whether it is part of a machine language program or data.

In this examination mode, the (↓) increments the address by one.

Word Mode

Get back into the command level and type:

W (ENTER)

Look at the same memory. Press the (↓) key a few times. The numbers are the same, but you are seeing them two bytes or one word at a time.

Here, the (↓) increments the address by two.

ASCII Mode

From the command level, type:

A (ENTER)

ZBUG is now assuming that the contents of each memory location is an ASCII code. If the "code" is between 21 and 7F (hexadecimal), ZBUG displays the character it represents. Otherwise, it displays nothing.

Examine the locations beginning with C056. These locations contain the Editor-Assembler + display heading.

Note: ZBUG will also display the A1 through FF as ASCII characters. However, they are not the true characters which these codes represent.

Here, the (↓) increments the address by one.

Mnemonic Mode

This is the default mode. Unless you ask for some other mode, as we have been doing, you will be in the default mode. To return to it, get in the command level and type:

M (ENTER)

Look at the locations beginning at C000 again. You'll see the same instructions you saw at the beginning of this chapter:

```
C000/            LDA #6
C002/            STA >0FF
etc.
```

In this mode, ZBUG assumes you're examining a machine language program. It examines memory from one to five bytes at a time by "disassembling" the numbers into the mnemonics they represent. The number 8606 (from locations C000 and C001) has been disassembled into LDA #6; B700FF (from locations C002, C003, and C004) into STA >FF; etc.

Begin the disassembly at a different byte. Type (BREAK) C001/ and press the (↓) several times. You will see a different disassembly:

```
C001/            ROR< 0B7
C003/            NEG< 0FF
etc.
```

The contents of memory have not changed. ZBUG has, however, interpreted them differently. The number 06B7 (from locations C001 and C002) has been disassembled into ROR<0B7; 00FF (from locations C003 and C004) has been disassembled into NEG<0FF; etc.

To see the program correctly, you must be sure you are beginning on the correct byte. Sometimes, several bytes will contain ??. This means ZBUG can't figure out what instruction is in that byte and is possibly disassembling from the wrong point. Unfortunately, though, the only sure way of knowing if you're on the right byte is by knowing where the program starts.

Changing Memory

As you look at the contents of memory locations, notice that the cursor is to the right. This allows you to change the contents of that location. After typing the new contents, press (ENTER) or (↓) and the change will be made.

For an example of changing memory, we'll open a location in Random Access Memory (RAM). Up to now, we've

only been examining locations in Read Only Memory (ROM) which we can't change. Get into the byte mode and open location 10AA by typing:

(BREAK) B (ENTER)
10AA/

Note that the cursor is to the right. Type:

1 (ENTER)

and the location now contains a 1. You can accomplish the same thing by typing:

10AA/

and then:

DD (↓)

which changes the contents to DD and allows you to change the next location. (Press (↓) to see that the change has been made.)

The size of the changes you make will depend on the examination mode you are in. In byte mode, you will change one byte only and can type one or two digits.

In the word mode, you will be changing one word at a time. Any one, two, three or four digit number you type will be the new value of the word.

If you happen to type a number which is also the name of one of the 6809 registers (A,B,D,CC,DP,X,Y,U,S,PC), ZBUG will assume it's a register and give you an "EXPRESSION ERROR." To avoid this confusion, type a leading zero (0A,0B,etc.).

To change memory in the ASCII mode, use an apostrophe before the new letter. For example, to write the letter A in memory at location 0000, type:

A (ENTER)

to go into ASCII examination mode, type:

0000/

to open that location and type:

'A (↓)

to change it. Typing the (↓) will assure you that the location contains the letter A.

If you are in mnemonic mode, you are expected to change one to five bytes of memory depending on the length of the particular instruction. Things get just a bit complex in mnemonic mode because you can't use mnemonic assembly language instructions. You must use the op code equivalent instead.

For example, get into the mnemonic mode and open location 1000. Type:

M (ENTER)
1000/

To change this instruction, type:

06 (ENTER)

Now location 1000 contains the op code for the LDA instruction. Open location 1001:

1001 /

and insert 06, the operand:

06 (ENTER)

Upon examining location 1000 again, you'll see it now contains a LDA #6 instruction.

Exploring the Computer's Memory

You are now invited to examine each section of memory using ZBUG commands to change examination modes. Use the Memory Map in *Appendix E*.

The following activities will allow you to become familiar with the Editor so don't be afraid to try commands or change memory. You can restore anything you alter by simply turning the computer OFF and ON again.

3/Writing the Program

To write assembly language programs, you will use the Editor. You can enter it by powering-up, pressing RESET, or (from ZBUG) typing E (ENTER). The asterisk prompt tells you that the Editor is available for commands. We say you are "in" the Editor.

The Editor has quite an assortment of commands to assist you. To use any of them, you must be at command level, as you are now. You can return to this command level by pressing (BREAK).

Sample Programming Exercise

For those of you new to editor-assemblers, we're including this sample programming exercise. We'll be referring to it in our examples throughout the manual. If you've used other editor-assemblers, you may skip this exercise and begin reading about the Write command.

To get started, type:

    I (ENTER)

Even though you have not typed anything yet, the Editor thinks that you are inserting lines into an already existing, although empty, edit buffer.

The Editor will respond with a line number. This line number is for your convenience while in the Editor and will not affect the machine language program at all.

To insert a comment line, type an asterisk and comment away. For example, insert this line:

    00100 *THIS IS A COMMENT LINE (ENTER)

The Assembler will ignore comment lines. You may type as many of them as you wish to explain your program to passing humans without confusing the computer.

You may delete this line and start over by pressing (BREAK) to get back into the command level and then typing:

    D100 (ENTER)

To type a program line, you will use four fields: the symbol, command, operand and comment fields. You can tab from one field to the next by pressing the (↑) key.

Insert this program line, using the (↑) key to tab from one column to the next:

    00100   SYMBOL CMD   OPERAND COMMENT (ENTER)

The symbol, command and operand fields must be terminated by a tab, space or carriage return. The symbol may be up to six characters. The comment is optional. The maximum line length is 128 characters. Note that long lines will "wrap around" your screen to the next line.

Delete whatever lines you have in the edit buffer and insert the following sample program. You may omit the comments, if you like:

```
00100  START  LDA  #$0F9  LOAD ASCII CHAR
00110         LDX  #$500  BEGIN VIDEO MEM
00120  SCREEN STA  ,X+    PUT CHAR ON SCREEN
00130         CMPX #$5FF  SEE IF END VIDEO MEM
00140         BNE  SCREEN BRANCH IF NOT
00150  DONE   SWI
00160         END
```

This stores graphics character number F9 into video memory locations 500-5FF. The dollar symbol ($) indicates a hexadecimal number. Without this symbol, the Editor will assume the number is decimal. (Note that the Editor defaults to decimal, whereas ZBUG defaults to hexadecimal.)

A description of all the other symbols, as well as the 6809 instructions, are in Part Two, "6809 Programming Reference Section."

Write Command
W filename

To save the sample program to tape (before making any experimental changes), type:

    W SAMPLE (ENTER)

You will be prompted with "READY CASSETTE". When the recorder is ready to record (i.e., you have inserted a tape and pressed PLAY and RECORD), type (ENTER). Your program will be saved as a "TEXT" file.

If you don't give your file a name, the default name NONAME will be assigned. It is a good idea to use file-names, especially if you will be storing more than one file on a single tape. Filenames may be up to eight characters long and must begin with a letter of the alphabet.

We recommend that you make a copy of your program before executing it. An assembly language program is not nearly as forgiving as BASIC. Executing the program with even a very small bug might result in erasing the entire edit buffer. In less than a second, many hours of editing and trial assembly can be completely obliterated!

After writing the file, it is useful to verify the tape with the V command. This command verifies the checksum on the tape. This verification could save frustration when saving long programs. The V command is listed in *Appendix A*.

Load Command
L *filename*

To load the TEXT file from tape, type:

L SAMPLE (ENTER)

You will be prompted to get your cassette recorder ready. (Rewind the tape and press PLAY.) When you press (ENTER), the recorder will begin searching for a file named SAMPLE. If you just want the first file, or whatever file is next on the tape, you may omit the filename.

This command will load a TEXT file only. (You will use the BASIC CLOADM command to load your assembled CODE file.)

> *Note: The Editor does not automatically empty its buffer before a LOAD. If a program is currently in memory, the program being LOADed will be appended to the one in memory.*
>
> *This can be useful for chaining long programs. When the second file is loaded, simply renumber the file (i.e., N100, 100).*
>
> *If you do not desire this, empty the buffer before loading a new program (i.e., D#:*).

Print Command
P*range*

To print a line of the program on the screen, type:

P100 (ENTER)

To print more than one line, type:

P100:130 (ENTER)

Since the first line, last line, and current line are very often referred to, you can refer to them with a single character:

=   first line

*   last line
,   current line (the last line you printed or inserted)

To print the current line, type:

P, (ENTER)

To print the entire text of the sample program, type:

P=:* (ENTER)

This is the same as P100:160 (ENTER).

The colon separates the beginning and ending lines in a range of lines. Another way to specify a range of lines is with !. Type:

P#15 (ENTER)

and five lines of your program, beginning with the first one, will be printed on the screen.

To stop the listing, you may quickly type:

(SHIFT) @

To continue, press any key.

Printer Commands
H*range*
T*range*

If you have a printer, you can print your program with the H and T commands. Both are closely related to the P command.

H=:* (ENTER)

will print every line of the edit buffer to the printer. You will be prompted with:

PRINTER READY

and you should respond with (ENTER) when ready.

T100!6 (ENTER)

will print six lines, beginning with line 100, to the printer, but without the Editor-supplied line numbers.

Edit Command
E*line*

You can edit lines in the same way you edit Extended BASIC lines. For example, to edit line 100, type:

E100 (ENTER)

The new line 100 is below old line 100 ready to be changed.

Press the (SPACEBAR) to position the cursor just after START and type this insert subcommand:

IED (ENTER)

which inserts ED in the line.

All the edit subcommands are listed in *Appendix A*.

Delete Command
Drange

If you are using the sample program, be sure you have written it on tape before you experiment with this command. Type:

    D110:140 (ENTER)

Lines 110 through 140 are gone.

Insert Command
Istartline,increment

Type:

    I152,2 (ENTER)

You may now insert lines beginning with line 152. Each line will be incremented by 2. (The Editor will not allow you to accidently overwrite an existing line. When you get to line 160, it will give you an error message.)

Press (BREAK) to return to the command level and type:

    I170 (ENTER)

This allows you to begin inserting lines at the end of the program. Each line will again be incremented by 2, the last increment you used.

Type:

    (BREAK) I (ENTER)

The Editor will begin inserting at the current line.

On start-up, the Editor sets the current line to 100 and the increment to 10. You may use any line numbers between 0 and 63999.

Renumber Command
Nstartline,increment

Another command that helps with inserting lines between the lines is N (for reNumber). From the command level, type:

    N100,50 (ENTER)

Now the lines begin with line 100 and are all incremented by 50. This allows you much more room for inserting between lines.

Type:

    N (ENTER)

The current line is now the first line number.

Renumber now so we will all be together for the next instruction. Type:

    N100,10 (ENTER)

Replace Command
Rstartline,increment

The replace command is a variation of the insert command. Type:

    R100,3 (ENTER)

You may now replace line 100 with a new line and begin inserting lines using an increment of three.

Copy Command
Cstartline,range,increment

The copy command will save you a lot of typing by duplicating any part of your program to another location in the program.

To copy lines, type:

    C500,100:150,10 (ENTER)

This will copy the range of lines from 100 to 150 to a new location beginning at line 500, with a line increment of 10. An attempt to copy lines over each other will fail.

ZBUG Command

To exit the Editor and enter ZBUG, type:

    Z (ENTER)

A different prompt, the #, tells you that you are now in ZBUG.

To re-enter the editor from ZBUG, type the ZBUG command:

    E (ENTER)

If you print your program, you'll see that entering and exiting ZBUG did not change it.

BASIC Command

To enter BASIC from the Editor, type:

    Q (ENTER)

for Quit. To re-enter the Editor from BASIC, type:

    EXEC 49152 (ENTER)

or

    EXEC &HC000 (ENTER)

which is the same address in hexadecimal. This is the first address of the Editor. You must use the decimal form if you have a 4K computer.

Entering BASIC will empty your edit buffer. Re-entering the Editor will empty your BASIC buffer.

Hints on Writing Your Program:

- Copy short programs unreservedly from any legal source available to you. Then modify them one

step at a time to learn how different commands and addressing modes work. Try to make the program relocatable by using indexed, relative, and indirect addressing (described in Part II).

Try to write a long program as a series of short routines that share the same symbols. They will be easier to understand and debug. They can later be combined into longer routines.

Note: You can use the Editor to edit your BASIC programs, as well as assembly language programs. You might find this very useful since the EDTASM+ Editor is much more powerful than BASIC's.

4 / Assembling

The command to assemble your text program into machine-code is simple. Just type (from the Editor command level):

A FILENAME (ENTER)

If your program is in memory, you will be prompted with:

CASSETTE READY

and when you press (ENTER) your cassette recorder will start. You are assembling the object program on tape for use another time and place. The Assembler will display a listing to explain what it is doing. (See *Figure 1* for an explanation of the listing.)

While this is the simplest form of the assemble command, it is not the one you will use first. You will want to make absolutely sure the program works before you assemble it to tape.

There are several options called switches which you can use to assemble the program for trial purposes. You may use any combination of these switches. For example:

A/IM/WE
A/WE/LP/NS
A TEST/LP

are all acceptable assembler commands.

/WE
Wait On Errors Switch

You will normally want to use this switch. It causes the Assembler to stop each time it encounters an error in your program. Press any key to continue the listing.

/SS /NO /NS /NL /LP
Listing Switches

Use these switches if you want the assembler listing (illustrated in *Figure 1*) to appear differently:

/SS   Short screen listing
/NO   No object code in the listing
/NS   No symbol table in the listing
/NL   No listing at all
/LP   Listing printed on the printer

/IM
Assembling In Memory Switch

The program will be assembled in memory, not on tape. This is usually for a trial assembly.

Where in memory? Used with no other switches, the Assembler will store your program just after the symbol table which is just after the edit buffer:
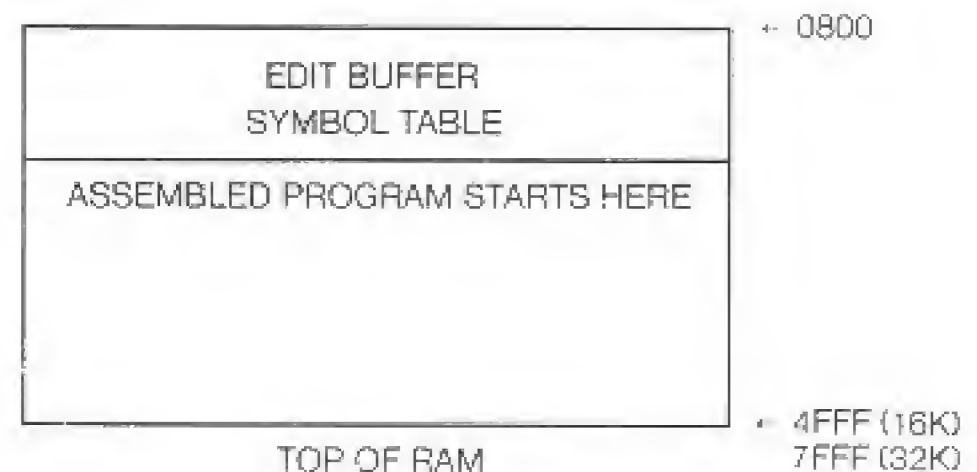
Figure 2. In Memory Assembly

The edit buffer contains your assembly language program. It begins at hexadecimal address 0800, and will vary in size depending on how long your program is.

The symbol table references all the symbols in your program and their corresponding values. Its size also varies depending on how many symbols your program has.

If you typed the sample program, you can try out an in-memory assembly. Make sure the program is in the Editor in its original form. Then, from the Editor command level, type:

A/IM (ENTER)

(If you want another look, type A/IM over again. You can pause the display with (SHIFT) (@) and continue with any key.)

Since this sample program uses START to label the beginning of the program, you can find its originating

```
0000 86    F9      00130 START
  LDA      #$0F9
0002 BE    0500       00110
  LDX      #$500
0005 A7    80      00120 SCREEN
  STA      ,X+
0007 8C    05FF    00130
  CMPX     #$5FF
000A 26    F9      00140
```

```
  BNE      SCREEN
000C 3F               00150 DONE
  SWI
           0000     00160
  END

00000 TOTAL ERRORS

DONE    000C
SCREEN  0005
START   0000
```
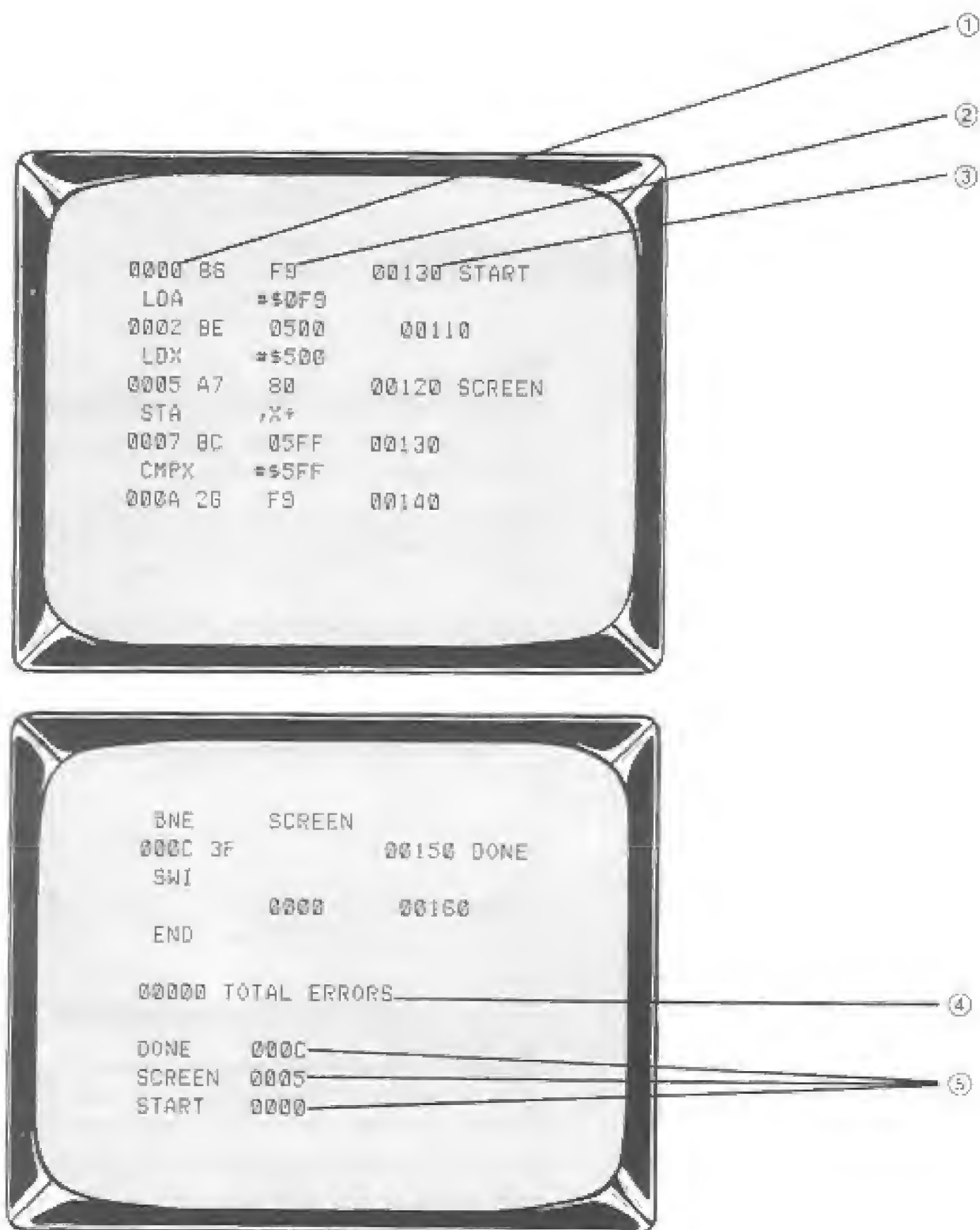
① The location in memory where the assembled code will be stored. In this example, the assembled code for LDA #$F9 will be stored at hexadecimal location 0000.

② The assembled code for the program line. 86F9 is the assembled code for LDA #$F9.

③ The program line.

④ The number of errors. If you have errors, you will want to assemble the program again with the /WE switch.

⑤ The symbols you used in your program and the memory locations they refer to.

Figure 1. Assembly Display Listing

14

address from the assembler listing. If you examine it with ZBUG, you'll see that it has been assembled into memory beginning between 0800 and 0900.

/AO
Absolute Origin Switch

This switch allows you to absolutely determine where in memory you want your assembled program to originate. To use it, you need to have an ORG instruction at the beginning of your program.

Insert this line at the beginning of the sample program:

```
00050        ORG  $3F00
```

Now type:

```
A/IM/AO
```

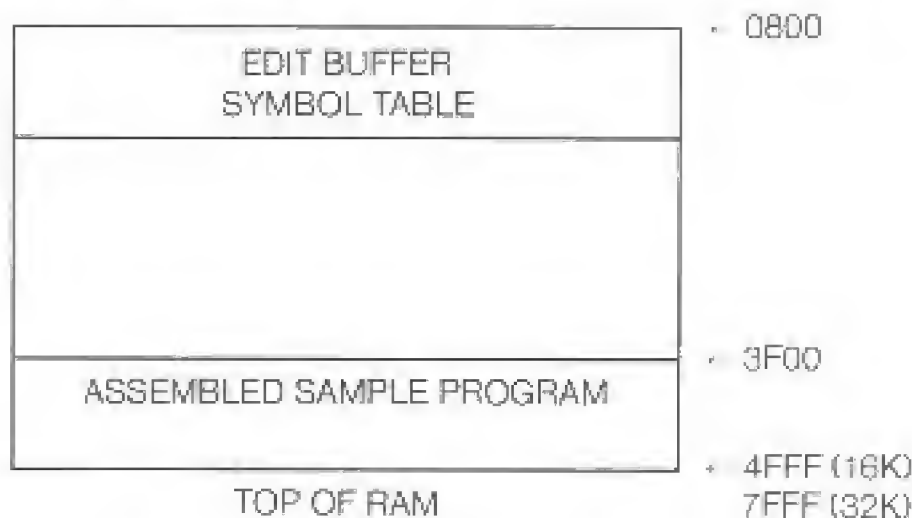If you use ZBUG, you'll see that your assembled program now begins at location 3F00:

EDIT BUFFER SYMBOL TABLE	· 0800
ASSEMBLED SAMPLE PROGRAM	· 3F00
TOP OF RAM	· 4FFF (16K) 7FFF (32K)

Figure 3. /AO In Memory Assembly

As you can see, the AO switch set the location of the assembled program only. It did not set the location of the edit buffer or the symbol table.

If your ORG instruction has not allowed enough room in memory for your program, you will get a BAD MEMORY error. The assembler cannot store your program beyond the top of RAM.

/MO
Manual Origin Switch

The manual origin switch offers you maximum control of in-memory assemblies. You can use it to assemble the program using the contents of these two memory addresses:

- USRORG (which contains the originating address of the assembled program)
- BEGTEMP (which contains 0600. This is the originating address of the edit buffer and the symbol table (which is 0800 minus 200.)

By manually changing the contents of USRORG and BEGTEMP, you'll be able to set the originating address of the edit buffer and symbol table as well as the executable program. Since this procedure is somewhat involved, not everyone will want to use the /MO switch.

To change the contents of these memory locations, you will need to get into ZBUG. Save the program you currently have in the Editor first. This procedure will destroy the contents of the edit buffer.

Then get into the ZBUG word mode by typing:

```
Z  (ENTER)
W  (ENTER)
```

and follow the procedures for setting USRORG or BEGTEMP (or both of them).

Setting USRORG

On start-up, 00FD points to the top of RAM. In this example, we'll change it to 2F00. Type:

```
FD/
2F00  (ENTER)
```

Now memory locations beginning with 2F00 are protected from EDTASM + and can be used for your assembled program.

Setting BEGTEMP

On start-up, 00FF points to 0600. In this example we'll change it to 2000. This will make room for high resolution graphics and data. Type:

```
FF/
2000  (ENTER)
```

The address you put in BEGTEMP must be:

- a "page boundary" (a hexadecimal number ending in 00)
- greater than 0600
- at least 300 bytes less than the contents of USRORG

Assembling the Program

To get back into the Editor, type:

```
GC006  (ENTER)
```

Load the sample program and, if you inserted an ORG instruction, delete it. Then type:

```
A/IM/MO  (ENTER)
```

This will assemble your program into the address you set for USRORG and BEGTEMP. If you followed our examples above, this command will assemble your program as follows:
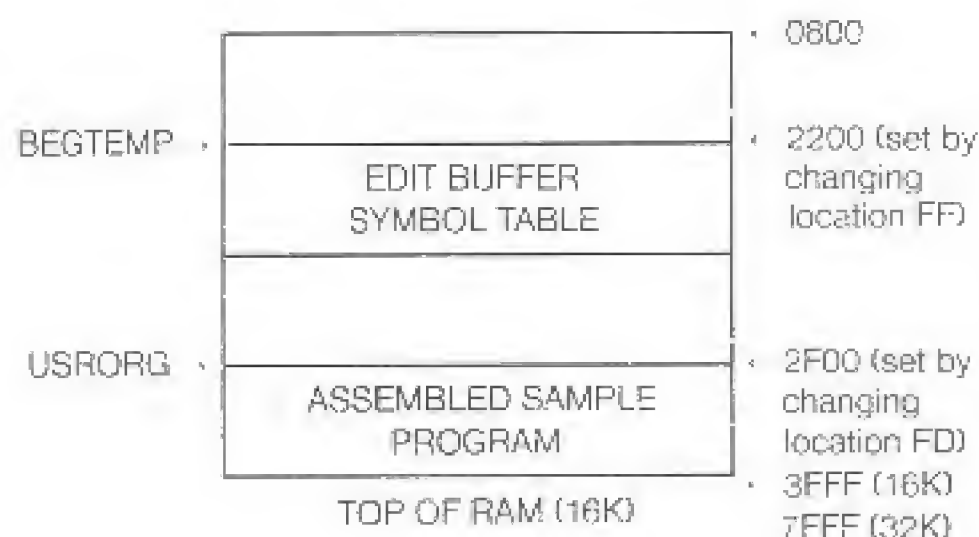
```
                                      ·  0800
          ┌─────────────────────┐
BEGTEMP · ├─────────────────────┤  ·  2200 (set by
          │    EDIT BUFFER      │        changing
          │   SYMBOL TABLE      │        location FF)
          │                     │
          │                     │
USRORG  · ├─────────────────────┤  ·  2F00 (set by
          │  ASSEMBLED SAMPLE   │        changing
          │      PROGRAM        │        location FD)
          └─────────────────────┘  ·  3FFF (16K)
            TOP OF RAM (16K)           7FFF (32K)
```

Figure 4. /MO In Memory Assembly

/NO
No Object Code Switch

Use this switch if you do not want to store any object code in memory or on tape.

Hints on Assembling

- Use a symbol to label the beginning of your program.

- Use the ORG instruction only when using the /AO switch. Used with /IM alone or /IM/MO, the ORG address will not be the program's originating address. The Assembler will use it to offset (add to) the loading address.

- The /WE switch is an excellent debugging tool. Use it to detect assembly errors before debugging the program.

- As your program library grows, it helps to use a different system of names to separate your TEXT, CODE, and BASIC files. For instance, you might want to use T, C, or B as the last letter of each file.

- If you would like to examine the edit buffer and symbol table after you assemble the program, use ZBUG to examine memory locations beginning with address 0800.

5 / Debugging with ZBUG

ZBUG has some very powerful tools for a trial run of your machine language program. You can use them to look at every register, every flag, and every memory location during every step of running the program.

Before reading any further, you might want to review the ZBUG commands you learned in *Chapter 1*. We will be using these commands in this chapter.

Sample Program Exercise

In this Chapter, we'll use the sample program to illustrate the debug commands. If you would like to use it and have not typed it in yet, see "Sample Programming Exercise" in *Chapter 2*.

Then insert an ORG $3F00 instruction at the beginning of the program (reinsert it, if you deleted it) and assemble the program using the /AO switch. See the discussion of the /AO switch in *Chapter 3* if you need help. Then enter ZBUG by typing "Z" from command level in the Editor.

Display Modes

In *Chapter 1*, we discussed four examination modes. ZBUG also has three display modes.

We'll examine each of these display modes from the mnemonic examination mode. If you're not in this mode, type M (ENTER).

Numeric Mode

Type:

N (ENTER)

and examine memory locations 3F00 through 3F0C, which contain your program. In the numeric mode, you will not see any of the symbols in your program (START, SCREEN, and DONE). All you see are numbers. For example, location 3F0A displays the instruction BNE 3F05 rather than BNE SCREEN.

Symbolic Mode

From the command level, type:

S (ENTER)

and examine your program again. ZBUG is displaying your entire program in terms of its symbols (START, SCREEN, and DONE). Examine the memory location containing the BNE SCREEN instruction and type:

;

The semicolon causes ZBUG to display the operand (SCREEN) as a number (3F05).

Half-Symbolic Mode

From the command level, type:

H (ENTER)

and examine the program. Now all the memory locations (on the left) are displayed as symbols, but the operands (on the right) are displayed as numbers.

Using Symbols to Examine Memory

Since ZBUG understands symbols, you can use them in your commands. For example, both of these commands open the same memory location (no matter which display mode you are in):

START/
3F00/

While either of these commands will get ZBUG to display your entire program:

T START DONE
T 3F00 3F0C

You can print this same listing on your printer by substituting TH for T.

Executing the Program

Before trying a trial run of the program, be sure you have a copy of it. As we've warned you, a small bug in it can destroy everything you have in memory.

You can run it from ZBUG using the G (Go) command followed by the program's start address. Type either of the following:

```
GSTART (ENTER)
G3F00 (ENTER)
```

and the program will execute, filling part of your screen with graphics character number F9. If it doesn't do this, the program probably has a "bug" which is what the rest of this chapter is about.

The 8 BRK @ 3F0C or 8 BRK @ DONE is ZBUG telling you that the program stopped executing at the SWI instruction located at 3F0C. ZBUG interprets your closing SWI instruction as the eighth or final "breakpoint" (discussed below).

Setting Breakpoints

If your program doesn't work properly, you might find it easier to debug it if you break it up into small units and run each unit separately. From the command level, type X followed by the address where you want execution to break.

We'll set a breakpoint at location 3F05, the first location containing the symbol SCREEN. To do this, type either of the following:

```
XSCREEN (ENTER)
X3F05 (ENTER)
```

Now type GSTART (ENTER) to execute the program. Each time execution breaks, type:

```
C (ENTER)
```

to continue. A graphics character will appear on the screen each time ZBUG executes the SCREEN loop. (The characters appear to be in a diagonal line because ZBUG scrolls to give you the breakpoint message.)

Type:

```
D (ENTER)
```

to display all the breakpoints you have set. Type:

```
C10 (ENTER)
```

and the tenth time ZBUG encounters that breakpoint, it halts execution. Type:

```
Y (ENTER)
```

This is the command to delete (Yank) a breakpoint. A breakpoint number after the Y will delete the breakpoint at that address. Used with no breakpoint number, ZBUG will delete all breakpoints.

You may set up to eight different breakpoints numbered 0 through 7. You may not set a breakpoint in a ROM routine.

Examining Registers and Flags

Type:

```
R (ENTER)
```

What you see are the contents of every register during this stage of program execution. (See Section II for a definition of all the 6809 registers and flags.)

Look at register CC (the Condition Code). Notice the letters to the right of it. These are the flags that are set in the CC register. The E, for example, means the E flag is set.

Type:

```
X/
```

and ZBUG displays only the contents of the X register. You can change this in the same way you change the contents of memory. Type:

```
0 (ENTER)
```

and the X register now contains a zero.

Stepping Through the Program

Type:

```
3F00,        Note the comma!
```

LDX #$500 is the next instruction to be executed. The first instruction, LDA #$FD, has just been executed. Type:

```
R (ENTER)
```

and you'll see this instruction has loaded register A with F9. To see the next instruction (LDX #$500) executed, type:

```
,        (Simply a comma)
```

You may continue single stepping through the program, examining the registers at will, until you reach the end. If you do manage to get to SWI, the last instruction, ZBUG will print:

```
CAN'T CONTINUE
```

which means it has reached the final step in the program. (SWI causes ZBUG to stop execution. If you omit SWI from your program, ZBUG will continue executing memory.)

Transferring a Block of Memory

Type:

```
U 3F00 0000 G (ENTER)
```

Now the first six bytes of your program have been copied to memory locations beginning with 0000.

Saving Memory on Tape

To save a block of memory from ZBUG, type:

```
P TEST 3F00 3F0C 3F00 (ENTER)
```

When the cassette is ready for recording, press (ENTER). This saves your program, beginning at memory location 3F00 and ending at 3F0C, on tape. The last number is where your program begins execution. In this case, this number is the same as the start address.

To load TEST back into ZBUG, type:

```
L TEST (ENTER)
```

Hints on Debugging

- *Don't expect your first program to work the first time. Have patience. Every programmer has bugs in his new programs, and debugging is a fact of life for all programmers, not just beginners.*

- *Be sure to make a copy of what you have in the edit buffer before executing the program. The edit buffer is not protected from machine language programs.*

6/Using the ZBUG Calculator

ZBUG has a built-in calculator that will perform arithmetic, relational, and logical operations. Furthermore, it allows you to interchangeably use three different numbering systems, ASCII characters, and symbols.

This Chapter contains many examples on how to use the calculator. Some of these examples require that you have the same sample program assembled in memory that we used in *Chapter 5*.

Numbering System Modes

ZBUG recognizes numbers in three numbering systems: hexadecimal (base 16), decimal (base 10), and octal (base 8).

Output Mode

The output mode determines which numbering system ZBUG will use to print or output numbers on the screen. From the ZBUG command level, type:

010 (ENTER)

and examine memory. The T at the end of each number stands for base 10. Type:

O8 (ENTER)

and you will see a Q at the end of each number. The numbers are all base 8. Type:

O16 (ENTER)

and you are now back in base 16, which is the default output mode.

Input Mode

You can change input modes in the same way you change output modes. For example, type:

I10 (ENTER)

Now ZBUG will interpret whatever number you input as a base 10 number. For example, if you are in this mode and type:

T 49152 49162 (ENTER)

ZBUG will show you memory locations 49152 (base 10) through 49162 (base 10). Note that what is printed on the screen is determined by the output mode, not the input mode.

You can use these special characters to "override" your input mode:

BASE	BEFORE NUMBER	AFTER NUMBER
Base 10	&	T
Base 16	S	H
Base 8	@	Q

Table 1. Special Input Mode Characters

For example, while still in the I10 mode, type:

T 49152 $C010 (ENTER)

The "$" overrides the I10 mode. ZBUG, therefore, interprets C010 as a hexadecimal number. As another example, get into the I16 mode and type:

T 49152T C010

Here, the "T" overrides the I16 mode. ZBUG interprets 49152 as decimal.

Operations

ZBUG will perform many different types of operations for you. For example, type:

C000+25T/

and ZBUG goes to memory location C019 (base 16), the sum of C000 (base 16) and 25 (base ten). If you simply want ZBUG to print the results of this calculation, type:

C000+25T=

On the following pages, we'll use the terms "operands," "operators," and "operation." An operation is any calculation you want ZBUG to solve. In this operation:

1+2=

"1" and "2" are the operands. "+" is the operator.

Operands

You may use any of these as operands:

1. ASCII characters

2. Symbols

3. Numbers (in either base 8, 10, or 16) — Please note that ZBUG will recognize integers (whole numbers) only

Examples:

    'A=

prints 41, the ASCII code for A.

    START=

prints the START address of the sample program. (It will print UNDEFINED SYMBOL if you don't have the sample program assembled in memory.)

    15Q=

prints the hexadecimal equivalent of octal 15.

If you would like your results printed in a different numbering system, use a different output mode. For example, get into the O10 mode and try all the above examples again.

Operators

You may use arithmetic, relational, or logical operators. (Get into the O16 mode for the following examples.)

Arithmetic Operators

Addition	+
Subtraction	-
Multiplication	*
Division	.DIV.
Modulus	.MOD.
Positive	+
Negative	-

Examples:

    DONE-START=

prints the length of the sample program (not including the SWI at the end).

    9.DIV.2=

prints 4. (ZBUG can perform only integer division.)

    9.MOD.2=

prints 1, the remainder of 9 divided by 2.

    1-2=

prints 0FFFF, 65535T, or 177777Q, depending on which output mode you are in. ZBUG will never calculate a negative number as a result. Instead, it uses a "number circle" which operates on modulus 10000 (hexadecimal):
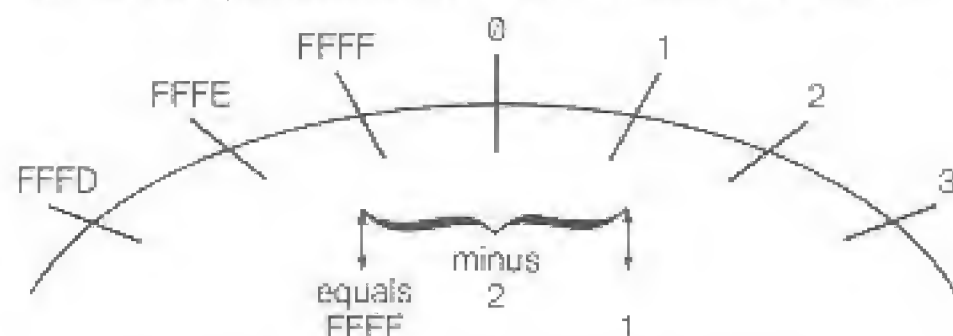

Figure 5. Number Circle Illustration of Memory

To understand this number circle, you can use the clock as an analogy. A clock operates on modulus 12 in the same way the ZBUG operates on modulus 10000. Therefore, on a clock. 1:00 minus 2 equals 11:00:
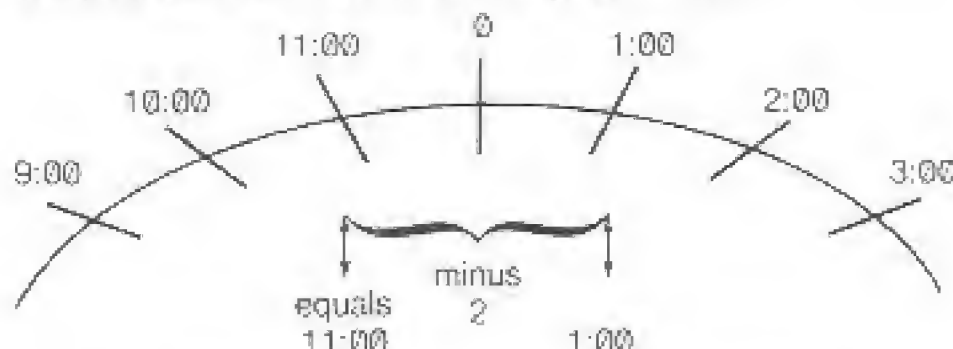

Figure 6. Number Circle Illustration of Clock

Relational Operators

Equals	.EQU.
Not Equal	.NEQ.

These operators determine whether a relationship is true or false.

Examples:

    5.EQU.5=

prints 0FFFF, since the relationship is true. (ZBUG will print 65535T in the O10 mode or 177777Q in the O8 mode.)

    5.NEQ.5=

prints 0, since the relationship is false.

Logical Operators

Shift	<
Logical AND	.AND.
Inclusive OR	.OR.
Exclusive XOR	.XOR.
Complement	.NOT.

Logical operators perform bit manipulation on binary numbers. To understand bit manipulations, see the 6809 assembly language book we referenced in the introduction.

Examples:

    10<2=

shifts 10 two bits to the left to equal 40. This is the same operation the 6809 ASL instruction performs.

    10<-2=

shifts 10 two bits to the right to equal 4. The ASR instruction also performs this operation.

    6.XOR.5=

prints 3, the Exclusive Or of 6 and 5. The 6809 EOR instruction performs this operation.

Complex Operations

ZBUG will calculate complex operations in this order:

```
      *  .DIV.  .MOD.  <
             .AND.
         .OR.  .XOR.
             +  -
         .EQU.  .NEQ.
```

You may use parentheses to change this order.

Examples:

   4+4.DIV.2=

The division is performed first.

   (4+4).DIV.2=

The addition is performed first.

   4*4.DIV.4=

The multiplication is performed first.

7/Running the Program From BASIC

The finished product of your labors is an assembled, debugged machine-code program. You can run this program directly from BASIC as either a stand-alone program or as a subroutine to your BASIC program.

The steps are:

From the Editor-Assembler:

1. Revise the program so that it will run as a routine and return to BASIC
2. Assemble the program on tape

From BASIC:

3. Load the assembled program with CLOADM
4. Execute the program
   - as a stand-alone program using EXEC, or
   - as a subroutine to your BASIC program using CLEAR and USR

1. Revising the Program

Before you can use the program from BASIC, you need to make a minor change to it. Change it to a routine which, after executing, will return to BASIC.

In our sample program, the next to the last instruction is:

```
SWI
```

Load the program into the Editor and change that instruction to:

```
RTS
```

Now the program is actually a routine which you can run from BASIC. (If you want to execute it again from ZBUG, you'll have to change RTS back to SWI or set a breakpoint before SWI and never execute it.)

So that your program is the same as ours, be sure that it has an ORG $3F00 instruction at the beginning of the program. This is the revised sample program.

```
        ORG     $3F00
START   LDA     #$0F9
        LDX     #$500
SCREEN  STA     ,X+
        CMPX    #$5FF
```

```
        BNE     SCREEN
DONE    RTS
        END
```

2. Assembling the Program

Once the program is revised, assemble it to tape with this command:

```
A SAMPLE (ENTER)
```

You are now finished with the Editor-Assembler, so you may start-up the Computer without the EDTASM+ ROM cartridge or enter BASIC with the Q command.

3. Loading the Program

To load the program, prepare your recorder and type:

```
CLOADM (ENTER)
```

Since we inserted an ORG $3F00 instruction in the sample program, you did not need to specify where in memory the program should be loaded. The program will be loaded at memory locations beginning with 3F00 (decimal 16128).

If your program does not have an ORG instruction, your CLOADM command will need to specify a loading address. CLOADM△16000 (ENTER), for example, would load the program into memory locations beginning with 16000.

4. Executing the Program

You can either execute the program as a stand-alone program or as a subroutine.

As a Stand-Alone Program

Type:

```
EXEC 16128 (ENTER)
```

The program will execute and return you to BASIC's OK prompt.

As a BASIC Subroutine

This is the most popular way to use machine language routines. When you need to do a task which is too slow or impossible in BASIC, you can call a machine-code subroutine. When the task is completed, it will return control to your BASIC program.

Type and RUN this BASIC program:

```
10 CLEAR 200, 16128
20 DEF USR0=16128
30 CLS
40 INPUT "PRESS <ENTER> WHEN READY"; A$
50 A=USR(0)
60 INPUT "WANT TO DO IT AGAIN"; A$
70 IF A$="YES" THEN 20
RUN (ENTER)
```

Normally BASIC can use any memory locations from decimal 1536 to the top of RAM. This means it could possibly overwrite your machine-code program. Line 10 CLEARs an area of memory from 16128 (which is hexadecimal 3F00) to the top of RAM, thereby restricting BASIC from using this area.

Line 20 defines the originating address of the machine-code program (USR) to be 16128. Line 50 calls the subroutine.

Passing Parameters

If you want to send some data to your machine-code program (we call this "passing a parameter"), you can substitute the "parameter" for the 0. For example:

```
A=USR(5)
```

will call the machine-code program and pass the param eter of 5 to it. To get this parameter, your machine-code program will need to have these two instructions:

```
INTCNV      EQU      $83ED
            JSR      [INTCNV]
```

which calls a routine called INTCNV. (INTCNV is locate in your BASIC ROM, along with other routines you ca use. All the BASIC ROM routines are listed in *Append E*.) INTCNV will get 5, the parameter in your USR state ment, and load it into the D register.

Your machine-code program can, in turn, return a param eter to your BASIC program by loading it in the D regist and then executing these instructions:

```
GIVABF      EQU      $B4F4
            JSR      [GIVABF]
```

GIVABF will set the variable in your USR statement, this case A, equal to the contents of the D register.

For more information on passing parameters, see th 6809 assembly language book we referenced in th introduction.

Note: to generate the [character, type (SHIFT) (←).
To generate the], type (SHIFT) (↓).

Hints and Tips

- *To save memory, use this formula to calculate the originating address of your program: top of RAM minus the length of the program (in bytes).*

8/6809 Assembly Language

This is a brief reference section on programming the 6809 microprocessor. It will not teach you assembly language programming.

Newcomers to assembly language programming will want to read:

Radio Shack Catalog No. 62-2077
by William Barden Jr.

Others, who want more information on the 6809 for technical applications, will want to read:

MC6809-MC6809E
8 Bit Microprocessor Programming Manual
Motorola, Inc.

The 6809 Microprocessor

The 6809 Microprocessor is produced by Motorola, Inc. It is an enhanced version of the MC6800 Microprocessor. Programs written on the 6800 are upwards compatible with the 6809.

Registers

The 6809 Processor contains nine temporary storage areas which you may use in your program:

REGISTER	SIZE	DESCRIPTION
A	1 byte	Accumulator
B	1 byte	Accumulator
D	2 bytes	Accumulator (a combination of A and B)
DP	1 byte	Direct Page
CC	1 byte	Condition Code
PC	2 bytes	Program Counter
X	2 bytes	Index
Y	2 bytes	Index
U	2 bytes	Stack Pointer
S	2 bytes	Stack Pointer

Table 2. 6809 Registers

The *A* and *B registers* are for manipulating data and doing arithmetic calculations. They can each hold one byte of data. If you like, you can address them as D, a single two byte register.

The *DP register* is for direct addressing. It will store the most significant byte of an address. This allows the Processor to directly access an address with the single, least significant byte.

The *X* and *Y registers* can each hold two bytes of data. You will use these registers primarily with indexed addressing.

The *PC register* stores the address of the next instruction to be executed.

The *U* and *S registers* can each hold a two byte address which points to an entire "stack" of memory. This address is one plus the top of the stack. For example, if the U register contains 0155, the stack begins with address 154 and continues downwards.

The processor automatically uses the S register to point to a stack of memory during subroutine calls and interrupts. The U register is solely for your own use. You can access either of these stacks with the PSH and PUL instructions or with indexed addressing.

The *CC register* is for testing conditions and setting interrupts. It is divided into eight "flags." Many 6809 operations will "set" or "clear" one or more of these flags. Other operations will test to see whether a certain flag is set or clear. This is the meaning of each flag, if set:

C (Carry), bit 0 — an 8-bit arithmetic operation caused a carry or borrow from bit 7.

V (Overflow), bit 1 — an arithmetic operation caused a signed overflow.

Z (Zero), bit 2 — the result of the previous operation is zero.

N (Negative), bit 3 — the result of the previous operation is a negative number.

I (Interrupt Request Mask), bit 4 — any requests for interrupts will be disabled.

H (Half Carry), bit 5 — an 8-bit addition operation caused a carry from bit 3.

F (Fast Interrupt Request Mask), bit 6 — any requests for fast interrupts will be disabled.

E (Entire Flag), bit 7 — all the registers were stacked during the last interrupt stacking operation. (If clear, only the PC and CC registers were stacked).

The Assembly Language Program

You may use four fields in an assembly language instruction: symbol, command, operand, comment. In this instruction:

```
START      LDA      #$F9      GETS CHAR
```

START is the symbol. LDA is the command. #$F9 is the operand (we will discuss the meaning of the # and $ signs later). GETS CHAR is the comment.

The comment is purely for your convenience. It is ignored by the Assembler.

The Symbol

You can use symbols to define memory addresses or data. The above instruction uses START to define its memory address. Once defined, you can use START as an operand in other instructions. For example:

```
BNE      START
```

branches to the memory address defined by START.

The Assembler stores all the symbols, along with the addresses or data they define, in a "symbol table" rather than as part of the "executable program."

The Command

The command may be either: a "pseudo-operation," or a 6809 instruction.

Pseudo-operations control various functions of the Assembler itself, such as defining labels, telling the Assembler where to store the executable program, or storing data in memory. They are not translated into 6809 machine-code and are not stored with the executable program. For example:

```
NAME      EQU      $43
```

defines the symbol NAME as 43. This information is stored in the symbol table.

```
ORG      $3000
```

tells the Assembler to begin the executable program at address 3000.

```
SYMBOL      FCB      $6
```

stores 6 in the current memory address and labels this address SYMBOL. SYMBOL and its corresponding address are stored in the symbol table.

6809 instructions tell the Microprocessor to carry out an operation. They are translated into 6809 machine-code as "op codes" and stored with the executable program. For example:

```
CLRA
```

tells the Processor to clear the A register. The Assembler translates this into op-code number 4F and stores it with the executable program.

All the pseudo-operations and 6809 instructions are listed at the end of this section.

The Operand

The operand allows you to specify a memory address or data. For example:

```
LDD      #$3000
```

loads register D with 3000. The operand, #$3000, specifies a data constant.

The $ sign indicates that 3000 is a hexadecimal, rather than decimal number. You must specify hexadecimal and octal numbers with:

BASE	BEFORE NUMBER	AFTER NUMBER
HEXADECIMAL	$	H
OCTAL	@	Q

Table 3. Hexadecimal and Octal Operands

For example, the Assembler interprets 17 as decimal 17; $17 as hexadecimal 17; and 17Q as octal 17.

The Assembler treats the operand as part of the 6809 instruction. It stores the operand with the executable program.

Addressing Modes

In the above example, we used the # sign to tell the Assembler and the Processor to interpret 3000 as data. We can specify a different mode of interpretation by omitting the # sign:

```
LDD      $3000
```

which interprets 3000 as an address. The Processor will then load D with the data contained in address 3000 and 3001.

Each of the 6809 operations allow you to use one to six addressing modes. These addressing modes tell you whether an operand is required to carry out the operation and which mode the Assembler and the Processor will use in interpreting the operand.

1. Inherent Addressing

There is no operand, since the instruction doesn't require one. For example:

```
            SWI
```

interrupts software. (No operand required.)

```
            CLRA
```

clears register A. Again, no operand is required. The A register is part of the instruction.

2. Immediate Addressing

The operand is *data*. You must use the # sign to specify this mode. For example:

```
            ADDA        #$30
```

adds the value 30 to the contents of the A register.

```
DATA        EQU         $8004
            LDX         #DATA
```

loads the value 8004 into the X register.

```
            CMPX        #$1234
```

compares the contents of register X with the value 1234.

3. Extended Addressing

The operand is an *address*. This is the default mode of all operands. (Exception: if the first byte of the operand is identical to the direct page, which is 00 on start-up, it will be directly addressed. This is an automatic function of the Assembler and the Processor. You do not need to be concerned with it if you're a beginner.) For example:

```
            JSR         $1234
```

jumps to address 1234.

```
SPOT        EQU         $1234
            STA         SPOT
```

stores the contents of register A in address 1234.

If the instruction calls for data, the operand contains the *address* where the *data* is stored.

```
            LDA         $1234
```

does not load register A with 1234. The Processor will load A with whatever data is in address 1234. If 06 is stored in address 1234, register A is loaded with 06.

```
            ADDA        $1234
```

adds whatever data is stored in address 1234 to the contents of register A.

```
            LDD         $1234
```

loads D, a two-byte register, with the data stored in memory locations 1234 and 1235.

You can use the > sign, which is the sign for extended addressing, to force this mode. (See "Direct Addressing").

Extended Indirect

The operand is an *address* of an *address*. This is a variation of the extended addressing mode. The [ ] signs specify it. (Use [SHIFT] (↓) to produce the [sign and [SHIFT] (↑) to produce the] sign.)

In understanding this mode, think of a treasure hunt game. The first instruction, "Look in the clock." The clock contains the second instruction, "Look in the refrigerator."

Examples:

```
            JSR         [$1234]
```

Jumps to the address that is contained in addresses 1234 and 1235. If 1234 contains 06 and 1235 contains 11, the effective address is 0611. The program will jump to 0611.

```
SPOT        EQU         $1234
            STA         [SPOT]
```

stores the contents of register A in the address contained in addresses 1234 and 1235.

```
            LDD         [$1234]
```

loads D with the data stored in the address stored in addresses 1234 and 1235.

This is a good mode of addressing to use when calling ROM routines. For example, the entry address of the POLCAT routine is contained in address A000. Therefore, you can call it with these instructions:

```
POLCAT      EQU         $A000
            JSR         [POLCAT]
```

If a new version of ROM puts the entry point in a different address, your program will work without any changes.

4. Indexed Addressing

The operand is an *index register* which points to an *address*. The *index register* may be any of the two byte registers, including PC. It may be augmented by:

- a constant or register offset
- an autoincrement or autodecrement of 1 or 2

The comma (,) indicates indexed addressing.

As an example, we'll first load X, a two byte register, with 1234:

```
            LDX         #$1234
```

We can now access address 1234 through indexed addressing. This instruction:

```
            STA         ,X
```

stores the contents of A in address 1234.

```
            STA         3,X
```

stores the contents of A in address 1237, which is 1234 – 3. (3 is a constant offset.)

```
SYMBOL     EQU        $4
           STA        SYMBOL,X
```

stores the contents of A in address 1238, which is 1234 + SYMBOL. (SYMBOL is a constant offset.)

```
           LDB        #$5
           STA        B,X
```

stores the contents of A in address 1239 which is 1234 + the contents of B. (B is a register offset. You may use either of the accumulator registers as a register offset.)

```
           STA        ,X+
```

This instruction does two tasks: (1) stores A's contents in address 1234 (the contents of X) and then (2) increments X's contents by one, so that X will contain 1235.

```
           STA        ,X++
```

(1) stores A's contents in address 1235 (the current contents of X) and then (2) increments X's contents by two to equal 1237.

```
           STA        ,--X
```

(1) decrements the current contents of X by two to equal 1235 (1237 – 2) and then (2) stores A's contents in address 1235.

As we said above, you can use PC as an index register. In this form of addressing, called program counter relative, the offset is interpreted differently. For example:

```
SYMBOL     FCB        0
           LDA        SYMBOL,PCR
```

When this program is assembled, the Assembler SUB-TRACTS the contents of the PC register from the offset:

```
           LDA        SYMBOL-PCR,PCR
```

When it is executed, the Processor ADDS the contents of the PC register to the offset. This causes A to be loaded with SYMBOL.

This appears to be the same as extended addressing. However, by using program counter relative addressing, the resulting machine-code program is completely relocatable.

Indexed Indirect Addressing

The operand is an *index register* which points to an *address* of an *address*. This is a variation of indexed addressing. For example, assuming that:

• the X register contains 1234
• address 1234 contains 11
• address 1235 contains 23

• address 1123 contains 64

this instruction:

```
           LDA        [,X]
```

loads A with 64. (The X register points to the addresses of the address. This address is storing 64, the required data.)

```
           STA        [,X]
```

stores the contents of A in address 1123. (The X register points to the addresses, 1234 and 1235, of the effective address, 1123.).

5. Relative Addressing

The Processor interprets the operand as a *relative address*. There is no sign to indicate this mode. The Processor automatically uses it for all branching instructions.

For example, if this instruction is located at address 0580:

```
           BRA        $0585
```

The Processor will convert $0600 to a relative branch of +5 (0600 – 0580).

As we said above, the Processor automatically uses this mode on all branching instructions. It is invisible to you unless you get a BYTE OVERFLOW error, which we'll discuss below. Because the Processor uses this mode, you can relocate your program in memory without changing any of the branching instructions.

The BYTE OVERFLOW error means that the relative branch is outside the range of –128 to +127. You will have to use a long branching instruction instead. For example:

```
           LBRA       $0600
```

allows a relative branching range of –32768 to +32767.

6. Direct Addressing

In this mode, the operand is *half of an address*. The other half of the address is the contents of the DP register:

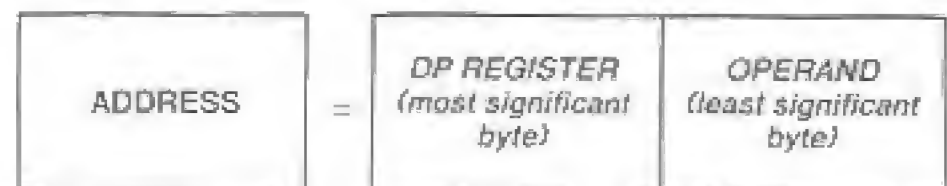| ADDRESS	=	DP REGISTER (most significant byte)	OPERAND (least significant byte)

Figure 7. Direct Addressing

The Assembler and the Processor use this mode automatically whenever they approach an operand whose first byte is what they assume to be a "direct page" (the contents of the DP register). Until you change the direct page, they both assume it is 00.

For example, both of these instructions:

```
JSR          $0015
JSR          $15
```

cause a jump to address 0015. In both cases, the Assembler uses only 15 as the operand, not 00. When the Processor executes them, it will get the 00 portion from the DP register and combine it with 15. (On start-up, DP contains 0, as do all the other registers.)

Because of direct addressing, all operands beginning with 00, the direct page, consume less room in memory and run quicker. If most of your operands begin with 12, you might want to make 12 the direct page.

To do this you first need to tell the Assembler what you are doing by putting a SETDP pseudo-operation in your program:

```
SETDP        $12
```

This tells the Assembler to drop the 12 from all operands beginning with 12. That is, the Assembler will assemble the operand "1234" as simply "34."

Then you must load the DP register with 12. Since you can use LD only with the accumulator registers, you will have to load DP in a round-about manner:

```
LDB          #$12
TFR          B,DP
```

Now the direct page is 12, rather than 00. The Processor will execute all operands beginning with 12 (rather than 00) in an efficient, direct manner.

The Assembler uses direct addressing on all operands whose first byte is the same as the direct page. You can be sure that the Assembler uses it or help document your program by using the < sign, which is the sign for direct addressing. For example, if the direct page is 12:

```
JSR          <$15
```

jumps to address 1215. This instruction documents that the Processor will use direct addressing.

Likewise, you might want to use the > sign to force extended addressing. For example:

```
JSR          >$1215
```

jumps to address 1215. The Assembler and Processor use both bytes of the operand.

9/Assembler Pseudo Operations

This is a listing of all the pseudo operations and the syntaxes you should use in typing them. Addressing modes do not apply to pseudo operations.

Definition of Terms

symbol

any string one to six characters long, typed in the symbol field.

expression

any expression typed in the operand field. See Appendix C, ZBUG commands, for a definition of valid expressions.

Pseudo Operations

END

END *expression*

Tells the Assembler to quit assembling the program. You can use the optional *expression* to specify the start address of the program. For example:

```
END          $3F00
```

tells the Assembler to quit assembling the program and to store its start address, 3F00, on tape. When you CLOADM the program, you will not need to specify the start address.

EQU

symbol **EQU** *expression*

Equates a *symbol* to an *expression*. For example:

```
LOOP1        EQU          $3F00
```

causes LOOP 1 to equal $3F00. You may use LOOP1 as data or an address.

EQU is helpful for setting the values of constants. You may use it anywhere in your program.

FCB

symbol **FCB** *expression*

Stores an *expression* into memory at the current address. The *symbol* is optional. The *expression* may be

one byte long. For example:

```
DATA         FCB          $33
```

stores 33 in address DATA.

```
DATA2        FCB          $33+COUNT
```

stores 33 + COUNT in address DATA2.

FCC

symbol **FCC** *delimiter string delimiter*

Stores an ASCII *string* into memory beginning with the current address. The *symbol* is optional. The *delimiter* may be any character. For example:

```
TABLE        FCC          /THIS IS A STRING/
```

writes the ASCII codes for THIS IS A STRING in memory locations beginning with TABLE.

FDB

symbol **FDB** *expression*

Stores an *expression* into memory beginning at the current address. The *symbol* is optional. The *expression* can be two bytes long. For example:

```
DATA         FDB          $3322
```

stores 3322 in address DATA and DATA + 1.

ORG

ORG *expression*

tells the Assembler to originate the program beginning with *expression*. For example:

```
ORG          $3F00
```

causes the assembler to begin assembling the program at address $3F00.

You may put more than one ORG command in a program. When the Assembler arrives at the new ORG command, it will begin locating program code at the new *expression*.

RMB

RMB *expression*

Reserves *expression* bytes of memory for data. For example:

```
DATA         RMB          $06
```

reserves 6 bytes for data beginning at address DATA.

SET

symbol SET *expression*

Sets *symbol* to be equal to *expression*. You may use SET to reset the *symbol* elsewhere in the program. For example:

```
SYMBOL         SET            $3500
```

sets SYMBOL equal to 3500. Later in the program, you may reset SYMBOL:

```
SYMBOL  SET  $4300
```

now SYMBOL equals 4300.

SETDP

SETDP *expression*

Tells the Assembler that the direct page will be *expression*. Example:

```
SETDP          $20
```

tells the Assembler to set the direct page to $20. You must also load the DP register with $20. See "Direct Addressing" for more information.

10/6809 Instruction Set

Definition of Terms

Source Forms:

This shows all the possible variations you can use with the instruction. *Table 4* gives the meaning of all the notations we use. The notations in italics represent values you can supply.

For example, the BEQ instruction has two source forms. BEQ *dd* allows you to use these instructions:

```
BEQ $0B     BEQ $FF     BEQ $A0
```

Whereas LBEQ DDDD allows you these:

```
LBEQ $C000     LBEQ $FFFF
```

Operation:

This uses shorthand notation to show exactly what the instruction does, step by step. The meaning of all the codes are also in *Table 4*.

For example, the BEQ operation does this:

> *"If, (but only if), the zero flag is set, branch to the location indicated by the program counter plus the value of the 8-bit offset."*

Condition Codes:

This shows which of the flags in the CC register are affected by the instruction, if any. As you'll note, BEQ does not set or clear any of the flags.

Description:

This is an overall description, in English, of what the instruction does.

Addressing Mode:

This tells you which addressing modes you may use with the instruction. BEQ allows only the Relative addressing mode.

ABBREVIATION	MEANING
ACCA or A	Accumulator A.
ACCB or B	Accumulator B.
ACCA:ACCB or D	Accumulator D.
ACCX	Either accumulator A or accumulator B.
CCR or CC	Condition code register.
DPR or DP	Direct page register.
EA	Effective address.
IFF	If and only if.
IX or X	Index register X.
IY or Y	Index register Y.
LSN	Least significant nibble.
M	Memory location.
MI	Memory immediate.
MSN	Most significant nibble.
PC	Program counter.
R	A register before the operation.
R'	A register after the operation.
TEMP	A temporary storage location.
xxH	Most significant byte of any location.
xxL	Least significant byte of any location
Sp or S	Hardware stack pointer.

ABBREVIATION	MEANING
Us or U	User stack pointer.
P	A memory location with immediate, direct, extended, and indexed addressing modes.
Q	A read-write-modify argument with direct, extended and indexed addressing modes.
( )	The data pointed to by the enclosed (16 bit address).
dd	8-bit branch offset.
DDDD	16-bit offset.
#	Immediate value follows.
$	Hexadecimal value follows.
[ ]	Indirection.
.	Indicates indexed addressing.
.	Is transferred to.
/	Boolean AND.
V	Boolean OR.
O	Boolean Exclusive OR (XOR).
—	Boolean NOT.
:	Concatination.
+	Arithmetic plus.
-	Arithmetic minus.
*.	Arithmetic multiply.

Table 4. Notations and Codes

Add Accumulator B into Index Register X

Source Form: ABX
Operation: IX' = IX + ACCB

Condition Codes: Not affected.
Description: Add the 8-bit unsigned value in accumulator B into index register X.
Addressing Mode: Inherent.

ABX

Add with Carry into Register

Source Forms: ADCA P; ADCB P
Operation: R' = R + M + C
Condition Codes:
  H — Set if a half-carry is generated; cleared otherwise.
  N — Set if the result is negative; cleared otherwise.
  Z — Set if the result is zero; cleared otherwise.
  V — Set if an overflow is generated; cleared otherwise.
  C — Set if a carry is generated; cleared otherwise.
Description: Adds the contents of the C (carry) bit and the memory byte into an 8-bit accumulator.
Addressing Modes: Immediate; Extended; Direct; Indexed.

ADC

Add Memory into Register

Source Forms: ADDA P; ADDB P
Operation: R' = R + M
Condition Codes:
  H — Set if a half-carry is generated; cleared otherwise.
  N — Set if the result is negative; cleared otherwise.
  Z — Set if the result is zero; cleared otherwise.
  V — Set if an overflow is generated; cleared otherwise.
  C — Set if a carry is generated; cleared otherwise.
Description: Adds the memory byte into an 8-bit accumulator.
Addressing Modes: Immediate; Extended; Direct; Indexed.

ADD
(8-Bit)

Add Memory into Register

Source Form: ADDD P
Operation: R' = R + M:M + 1
Condition Codes:
  H — Not affected
  N — Set if the result is negative; cleared otherwise.
  Z — Set if the result is zero; cleared otherwise.
  V — Set if an overflow is generated; cleared otherwise.
  C — Set if a carry is generated; cleared otherwise.
Description: Adds the 16-bit memory value into the 16-bit accumulator.
Addressing Modes: Immediate; Extended; Direct; Indexed

ADD
(16-Bit)

Logical AND Memory into Register

Source Forms: ANDA P; ANDB P
Operation: R' = R ∧ M
Condition Codes:
  H — Not affected.
  N — Set if the result is negative; cleared otherwise
  Z — Set if the result is zero; cleared otherwise.
  V — Always cleared
  C — Not affected
Description: Performs the logical AND operation between the contents of an accumulator and the contents of memory location M and the result is stored in the accumulator.
Addressing Modes: Immediate; Extended; Direct; Indexed.

AND

Logical AND Immediate Memory into Condition Code Register

Source Form: ANDCC #xx
Operation: R' = R ∧ MI
Condition Codes: Affected according to the operation

Description: Performs a logical AND between the condition code register and the immediate byte specified in the instruction and places the result in the condition code register.
Addressing Mode: Immediate.

AND

Arithmetic Shift Left

Source Forms: ASL Q; ASLA; ASLB
Operation: C ← [| | | | | | | |] ← 0
            b7   •   b0
Condition Codes:
  H — Undefined
  N — Set if the result is negative; cleared otherwise.
  Z — Set if the result is zero; cleared otherwise.
  V — Loaded with the result of the exclusive OR of bits six and seven of the original operand
  C — Loaded with bit seven of the original operand.
Description: Shifts all bits of the operand one place to the left. Bit zero is loaded with a zero. Bit seven is shifted into the C (carry) bit.
Addressing Modes: Inherent; Extended; Direct; Indexed.

ASL

ASR — Arithmetic Shift Right

Source Forms: ASR Q; ASRA; ASRB

Operation:

b7   b0

Condition Codes:
H   Undefined.
N   Set if the result is negative; cleared otherwise.
Z   Set if the result is zero; cleared otherwise.
V   Not affected.
C   Loaded with bit zero of the original operand.

Description: Shifts all bits of the operand one place to the right. Bit seven is held constant. Bit zero is shifted into the C (carry) bit.

Addressing Modes: Inherent; Extended; Direct; Indexed.

BCC — Branch on Carry Clear

Source Forms: BCC dd; LBCC DDDD

Operation:
    TEMP← MI
    IFF C = 0 then PC'← PC + TEMP

Condition Codes: Not affected.
Description: Tests the state of the C (carry) bit and causes a branch if it is clear.
Addressing Mode: Relative.
Comments: Equivalent to BHS dd; LBHS DDDD.

BCS — Branch on Carry Set

Source Forms: BCS dd; LBCS DDDD

Operation:
    TEMP← MI
    IFF C = 1 then PC'← PC + TEMP

Condition Codes: Not affected.
Description: Tests the state of the C (carry) bit and causes a branch if it is set.
Addressing Mode: Relative.
Comments: Equivalent to BLO dd; LBLO DDDD.

BEQ — Branch on Equal

Source Forms: BEQ dd; LBEQ DDDD

Operation:
    TEMP← MI
    IFF Z = 1 then PC'← PC + TEMP

Condition Codes: Not affected.

Description: Tests the state of the Z (zero) bit and causes a branch if it is set. When used after a subtract or compare operation, this instruction will branch if the compared values, signed or unsigned, were exactly the same.
Addressing Mode: Relative.

BGE — Branch on Greater than or Equal to Zero

Source Forms: BGE dd; LBGE DDDD

Operation:
    TEMP← MI
    IFF [N ⊕ V] = 0 then PC'← PC + TEMP

Condition Codes: Not affected.

Description: Causes a branch if the N (negative) bit and the V (overflow) bit are either both set or both clear. That is, branch if the sign of a valid twos complement result is, or would be, positive. When used after a subtract or compare operation on twos complement values, this instruction will branch if the register was greater than or equal to the memory operand.
Addressing Mode: Relative.

BGT — Branch on Greater

Source Forms: BGT dd; LBGT DDDD

Operation:
    TEMP← MI
    IFF Z ∧ [N ⊕ V] = 0 then PC'← PC + TEMP

Condition Codes: Not affected.
Description: Causes a branch if the N (negative) bit and V (overflow) bit are either both set or both clear and the Z (zero) bit is clear. In other words, branch if the sign of a valid twos complement result is, or would be, positive and not zero. When used after a subtract or compare operation on twos complement values, this instruction will branch if the register was greater than the memory operand.
Addressing Mode: Relative.

BHI — Branch if Higher

Source Forms: BHI dd; LBHI DDDD

Operation:
    TEMP← MI
    IFF [C ∨ Z] = 0 then PC'← PC + TEMP

Condition Codes: Not affected.
Description: Causes a branch if the previous operation caused neither a carry nor a zero result. When used after a subtract or compare operation on unsigned binary values, this instruction will branch if the register was higher than the memory operand.
Addressing Mode: Relative.
Comments: Generally not useful after INC/DEC, LD/TST, and TST/CLR/COM instructions.

Branch if Higher or Same

Source Forms: BHS dd; LBHS DDDD
Operation:
   TEMP ← MI
   IFF C = 0 then PC' ← PC + MI
Condition Codes: Not affected.
Description: Tests the state of the C (carry) bit and causes a branch if it is clear. When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was higher than or the same as the memory operand.
Addressing Mode: Relative.
Comments: This is a duplicate assembly-language mnemonic for the single machine instruction BCC. Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM instructions.

BHS

Bit Test

Source Form: BIT P
Operation: TEMP ← R ∧ M
Condition Codes:
   H — Not affected
   N — Set if the result is negative; cleared otherwise
   Z — Set if the result is zero; cleared otherwise.
   V — Always cleared
   C — Not affected.
Description: Performs the logical AND of the contents of accumulator A or B and the contents of memory location M and modifies the condition codes accordingly. The contents of accumulator A or B and memory location M are not affected.
Addressing Modes: Immediate; Extended; Direct; Indexed.

BIT

Branch on Less than or Equal to Zero

Source Forms: BLE dd; LBLE DDDD
Operation:
   TEMP ← MI
   IFF Z v (N ⊕ V) = 1 then PC' ← PC + TEMP
Condition Codes: Not affected.
Description: Causes a branch if the exclusive OR of the N (negative) and V (overflow) bits is 1 or if the Z (zero) bit is set. That is, branch if the sign of a valid twos complement result is, or would be, negative. When used after a subtract or compare operation on twos complement values, this instruction will branch if the register was less than or equal to the memory operand.
Addressing Mode: Relative.

BLE

Branch on Lower

Source Forms: BLO dd; LBLO DDDD
Operation:
   TEMP ← MI
   IFF C = 1 then PC' ← PC + TEMP
Condition Codes: Not affected.
Description: Tests the state of the C (carry) bit and causes a branch if it is set. When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was lower than the memory operand.
Addressing Mode: Relative.
Comments: This is a duplicate assembly-language mnemonic for the single machine instruction BCS. Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM instructions.

BLO

Branch on Lower or Same

Source Forms: BLS dd; LBLS DDDD
Operation:
   TEMP ← MI
   IFF (C v Z) = 1 then PC' ← PC + TEMP
Condition Codes: Not affected.
Description: Causes a branch if the previous operation caused either a carry or a zero result. When used after a subtract or compare operation on unsigned binary values, this instruction will branch if the register was lower than or the same as the memory operand.
Addressing Mode: Relative.
Comments: Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM instructions.

BLS

Branch on Less than Zero

Source Forms: BLT dd; LBLT DDDD
Operation:
   TEMP ← MI
   IFF (N ⊕ V) = 1 then PC' ← PC + TEMP
Condition Codes: Not affected.
Description: Causes a branch if either, but not both, of the N (negative) or V (overflow) bits is set. That is, branch if the sign of a valid twos complement result is, or would be, negative. When used after a subtract or compare operation on twos complement binary values, this instruction will branch if the register was less than the memory operand.
Addressing Mode: Relative.

BLT

Branch on Minus

Source Forms: BMI dd; LBMI DDDD
Operation:
   TEMP ← MI
   IFF N = 1 then PC' ← PC + TEMP
Condition Codes: Not affected.
Description: Tests the state of the N (negative) bit and causes a branch if set. That is, branch if the sign of the twos complement result is negative.
Addressing Mode: Relative
Comments: When used after an operation on signed binary values, this instruction will branch if the result is minus. It is generally preferred to use the LBLT instruction after signed operations.

BMI

BNE

Branch Not Equal

Source Forms: BNE dd; LBNE DDDD
Operation:
    TEMP← MI
    IFF Z=0 then PC'← PC + TEMP
Condition Codes: Not affected.

Description: Tests the state of the Z (zero) bit and causes a branch if it is clear. When used after a subtract or compare operation on any binary values, this instruction will branch if the register is, or would be, not equal to the memory operand.
Addressing Mode: Relative.

BPL

Branch on Plus

Source Forms: BPL dd; LBPL DDDD
Operation:
    TEMP← MI
    IFF N=0 then PC'← PC + TEMP
Condition Codes: Not affected.
Description: Tests the state of the N (negative) bit and

causes a branch if it is clear. That is, branch if the sign of the twos complement result is positive.
Addressing Mode: Relative.
Comments: When used after an operation on signed binary values, this instruction will branch if the result (possibly invalid) is positive. It is generally preferred to use the BGE instruction after signed operations.

BRA

Branch Always

Source Forms: BRA dd; LBRA DDDD
Operation:
    TEMP← MI
    PC'← PC + TEMP

Condition Codes: Not affected
Description: Causes an unconditional branch
Addressing Mode: Relative.

BRN

Branch Never

Source Forms: BRN dd; LBRN DDDD
Operation: TEMP←MI
Condition Codes: Not affected

Description: Does not cause a branch. This instruction is essentially a no operation, but has a bit pattern logically related to branch always.
Addressing Mode: Relative

BSR

Branch to Subroutine

Source Forms: BSR dd; LBSR DDDD
Operation:
    TEMP← MI
    SP'← SP-1, (SP)← PCL
    SP'← SP-1, (SP)← PCH
    PC'← PC + TEMP

Condition Codes: Not affected.
Description: The program counter is pushed onto the stack. The program counter is then loaded with the sum of the program counter and the offset.
Addressing Mode: Relative.
Comments: A return from subroutine (RTS) instruction is used to reverse this process and must be the last instruction executed in a subroutine

BVC

Branch on Overflow Clear

Source Forms: BVC dd; LBVC DDDD
Operation:
    TEMP← MI
    IFF V=0 then PC'← PC + TEMP
Condition Codes: Not affected

Description: Tests the state of the V (overflow) bit and causes a branch if it is clear. That is, branch if the twos complement result was valid. When used after an operation on twos complement binary values, this instruction will branch if there was no overflow.
Addressing Mode: Relative.

CMP
(8-Bit)

Compare Memory from Register

Source Forms: CMPA P; CMPB P
Operation: TEMP← R - M
Condition Codes:
    H — Undefined.
    N — Set if the result is negative; cleared otherwise.
    Z — Set if the result is zero; cleared otherwise.

    V   Set if an overflow is generated; cleared otherwise.
    C   Set if a borrow is generated; cleared otherwise.
Description: Compares the contents of memory location to the contents of the specified register and sets the appropriate condition codes. Neither memory location M nor the specified register is modified. The carry flag represents a borrow and is set to the inverse of the resulting binary carry
Addressing Modes: Immediate; Extended; Direct; Indexed

Compare Memory from Register

Source Forms: CMPD P; CMPX P; CMPY P; CMPU P; CMPS P

Operation: TEMP- R - M:M + 1

Condition Codes:
- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Set if an overflow is generated; cleared otherwise.
- C — Set if a borrow is generated; cleared otherwise.

Description: Compares the 16-bit contents of the concatenated memory locations M:M + 1 to the contents of the specified register and sets the appropriate condition codes. Neither the memory locations nor the specified register is modified unless autoincrement or autodecrement are used. The carry flag represents a borrow and is set to the inverse of the resulting binary carry.

Addressing Modes: Immediate; Extended; Direct; Indexed.

Complement

Source Forms: COM Q; COMA; COMB

Operation: M ← 0 + M

Condition Codes:
- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Always cleared.
- C — Always set.

Description: Replaces the contents of memory location M or accumulator A or B with its logical complement. When operating on unsigned values, only BEQ and BNE branches can be expected to behave properly following a COM instruction. When operating on twos complement values, all signed branches are available.

Addressing Modes: Inherent; Extended; Direct; Indexed.

Clear CC bits and Wait for Interrupt

Source Form: CWAI #$XX | E | F | H | I | N | Z | V | C |

Operation:
CCR ← CCR ∧ MI (Possibly clear masks)
Set E (entire state saved)
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
SP' ← SP - 1, (SP) ← USL
SP' ← SP - 1, (SP) ← USH
SP' ← SP - 1, (SP) ← IYL
SP' ← SP - 1, (SP) ← IYH
SP' ← SP - 1, (SP) ← IXL
SP' ← SP - 1, (SP) ← IXH
SP' ← SP - 1, (SP) ← DPR
SP' ← SP - 1, (SP) ← ACCB
SP' ← SP - 1, (SP) ← ACCA
SP' ← SP - 1, (SP) ← CCR

Condition Codes: Affected according to the operation.

Description: This instruction ANDs an immediate byte with the condition code register which may clear the interrupt mask bits I and F, stacks the entire machine state on the hardware stack and then looks for an interrupt. When a non-masked interrupt occurs, no further machine state information need be saved before vectoring to the interrupt handling routine. This instruction replaced the MC6800 CLI WAI sequence, but does not place the buses in a high-impedance state. A FIRQ (fast interrupt request) may enter its interrupt handler with its entire machine state saved. The RTI (return from interrupt) instruction will automatically return the entire machine state after testing the E (entire) bit of the recovered condition code register.

Addressing Mode: Immediate.

Comments: The following immediate values will have the following results:
FF — enable neither
EF — enable IRQ
BF — enable FIRQ
AF — enable both

Decimal Addition Adjust

Source Form: DAA

Operation: ACCA ← ACCA + CF (MSN):CF(LSN)

where CF is a Correction Factor, as follows: the CF for each nibble (BCD) digit is determined separately, and is either 6 or 0.

Least Significant Nibble
CF(LSN) = 6 IFF 1) C = 1
or 2) LSN > 9

Most Significant Nibble
CF(MSN) = 6 IFF 1) C = 1
or 2) MSN > 9
or 3) MSN > 8 and LSN > 9

Condition Codes:
- H — Not affected.
- N — Set if the result is negative; cleared otherwise.
- Z — Set if the result is zero; cleared otherwise.
- V — Undefined.
- C — Set if a carry is generated or if the carry bit was set before the operation; cleared otherwise.

Description: The sequence of a single-byte add instruction on accumulator A (either ADDA or ADCA) and a following decimal addition adjust instruction results in a BCD addition with an appropriate carry bit. Both values to be added must be in proper BCD form (each nibble such that: 0 ≤ nibble ≤ 9). Multiple-precision addition must add the carry generated by this decimal addition adjust into the next higher digit during the add operation (ADCA) immediately prior to the next decimal addition adjust.

Addressing Mode: Inherent.

DEC

Decrement

Source Forms: DEC Q; DECA; DECB
Operation: M' ← M - 1
Condition Codes:
- H   Not affected.
- N   Set if the result is negative; cleared otherwise.
- Z   Set if the result is zero; cleared otherwise.
- V   Set if the original operand was 10000000; cleared otherwise.

C — Not affected.
Description: Subtract one from the operand. The carry bit is not affected, thus allowing this instruction to be used as a loop counter in multiple-precision computations. When operating on unsigned values, only BEQ and BNE branches can be expected to behave consistently. When operating on twos complement values, all signed branches are available.
Addressing Modes: Inherent; Extended; Direct; Indexed.

EOR

Exclusive OR

Source Forms: EORA P; EORB P
Operation: R' ← R ⊕ M
Condition Codes:
- H   Not affected.
- N   Set if the result is negative; cleared otherwise.

Z   Set if the result is zero; cleared otherwise.
V   Always cleared.
C   Not affected
Description: The contents of memory location M is exclusive ORed into an 8-bit register.
Addressing Modes: Immediate; Extended; Direct; Indexed.

EXG

Exchange Registers

Source Form: EXG R1,R2
Operation: R1 ↔ R2
Condition Codes: Not affected (unless one of the registers is the condition code register)
Description: Exchanges data between two designated registers. Bits 3-0 of the postbyte define one register, while bits 7-4 define the other, as follows:

| 0000 – A:B | 1000 – A |
| 0001 – X | 1001 – B |

0010 – Y	1010 – CCR
0011 – US	1011 – DPR
0100 – SP	1100 – Undefined
0101 – PC	1101 – Undefined
0110 – Undefined	1110 – Undefined
0111 – Undefined	1111 – Undefined

Only like size registers may be exchanged (8-bit with 8-bit or 16-bit with 16-bit.)
Addressing Mode: Immediate.

INC

Increment

Source Forms: INC Q; INCA; INCB
Operation: M' ← M + 1
Condition Codes:
- H   Not affected.
- N   Set if the result is negative; cleared otherwise.
- Z   Set if the result is zero; cleared otherwise.
- V   Set if the original operand was 01111111; cleared otherwise

C — Not affected.
Description: Adds to the operand. The carry bit is not affected, thus allowing this instruction to be used as a loop counter in multiple-precision computations. When operating on unsigned values, only the BEQ and BNE branches can be expected to behave consistently. When operating on twos complement values, all signed branches are correctly available.
Addressing Modes: Inherent; Extended; Direct; Indexed

JMP

Jump

Source Form: JMP EA
Operation: PC ← EA
Condition Codes: Not affected.

Description: Program control is transferred to the effective address.
Addressing Modes: Extended; Direct; Indexed.

JSR

Jump to Subroutine

Source Form: JSR EA
Operation:
SP' ← SP - 1, (SP) ← PCL
SP' ← SP - 1, (SP) ← PCH
PC ← EA

Condition Codes: Not affected.
Description: Program control is transferred to the effective address after storing the return address on the hardware stack. A RTS instruction should be the last executed instruction of the subroutine.
Addressing Modes: Extended; Direct; Indexed.

LD (8-Bit)

Load Register from Memory

Source Forms: LDA P; LDB P
Operation: R' ← M
Condition Codes:
- H   Not affected.
- N   Set if the loaded data is negative; cleared otherwise.

Z — Set if the loaded data is zero; cleared otherwise
V — Always cleared.
C — Not affected.
Description: Loads the contents of memory location M into the designated register.
Addressing Modes: Immediate; Extended; Direct; Indexed.

Load Register from Memory

Source Forms: LDD *P*; LDX *P*; LDY *P*; LDS *P*; LDU *P*
Operation: R' - M:M + 1
Condition Codes:
  H — Not affected
  N — Set if the loaded data is negative; cleared otherwise

  Z — Set if the loaded data is zero; cleared otherwise.
  V — Always cleared.
  C — Not affected.
Description: Load the contents of the memory location M:M - 1 into the designated 16-bit register.
Addressing Modes: Immediate; Extended; Direct; Indexed.

Load Effective Address

Source Forms: LEAX, LEAY, LEAS, LEAU
Operation: R' - EA
Condition Codes:
  H — Not affected.
  N — Not affected
  Z — LEAX, LEAY: Set if the result is zero; cleared otherwise. LEAS, LEAU: Not affected.
  V — Not affected.
  C — Not affected.
Description: Calculates the effective address from the index addressing mode and places the address in an indexable register.

LEAX and LEAY affect the Z (zero) bit to allow use of these registers as counters and for MC6800 INX/DEX compatibility

LEAU and LEAS do not affect the Z bit to allow cleaning up the stack while returning the Z bit as a parameter to a calling

routine, and also for MC6800 INS/DES compatibility.
Addressing Mode: Indexed.
Comments: Due to the order in which effective addresses are calculated internally, the LEAX, X + 1 and LEAX,X + do not add 2 and 1 (respectively) to the X register, but instead leave the X register unchanged. This also applies to the Y, U, and S registers. For the expected results, use the faster instruction LEAX 2, X and LEAX 1, X.

Some examples of LEA instruction uses are given in the following table.

Instruction	Operation	Comment
LEAX 10, X	X - 10 - X	Adds 5-bit constant 10 to X.
LEAX 500, X	X - 500 - X	Adds 16-bit constant 500 to X.
LEAY A, Y	Y - A - Y	Adds 8-bit accumulator to Y.
LEAY D, Y	Y - D - Y	Adds 16-bit D accumulator to Y.
LEAU 10, U	U - 10 - U	Subtracts 10 from U.
LEAS - 10, S	S - 10 - S	Used to reserve area on stack.
LEAS 10, S	S + 10 - S	Used to 'clean up' stack.
LEAX 5, S	S + 5 - X	Transfers as well as adds.

Logical Shift Left

Source Forms: LSL *Q*; LSLA; LSLB

Operation: C - [| | | | | | | /] - 0
    b7   b0

Condition Codes:
  H — Undefined.
  N — Set if the result is negative; cleared otherwise.
  Z — Set if the result is zero; cleared otherwise.

  V — Loaded with the result of the exclusive OR of bits six and seven of the original operand.
  C — Loaded with bit seven of the original operand.
Description: Shifts all bits of accumulator A or B or memory location M one place to the left. Bit zero is loaded with a zero. Bit seven of accumulator A or B or memory location M is shifted into the C (carry) bit.
Addressing Modes: Inherent; Extended; Direct; Indexed.
Comments: This is a duplicate assembly-language mnemonic for the single machine instruction ASL.

Logical Shift Right

Source Forms: LSR *Q*; LSRA; LSRB

Operation: D - [| | | | | | |] - C
    b7   b0

Condition Codes:
  H — Not affected.

  N — Always cleared.
  Z — Set if the result is zero; cleared otherwise.
  V — Not affected.
  C — Loaded with bit zero of the original operand.
Description: Performs a logical shift right on the operand. Shifts a zero into bit seven and bit zero into the C (carry) bit.
Addressing Modes: Inherent; Extended; Direct; Indexed.

Multiply

Source Form: MUL
Operation: ACCA':ACCB' - ACCA × ACCB
Condition Codes:
  H — Not affected.
  N — Not affected.
  Z — Set if the result is zero; cleared otherwise.
  V — Not affected.

  C — Set if ACCB bit 7 of result is set; cleared otherwise.
Description: Multiply the unsigned binary numbers in the accumulators and place the result in both accumulators (ACCA contains the most-significant byte of the result). Unsigned multiply allows multiple-precision operations
Addressing Mode: Inherent.
Comments: The C (carry) bit allows rounding the most-significant byte through the sequence: MUL, ADCA #0.

NEG

Negate

Source Forms: NEG Q; NEGA; NEGB
Operation: M' - 0 - M
Condition Codes:
  H - Undefined.
  N - Set if the result is negative; cleared otherwise.
  Z - Set if the result is zero; cleared otherwise.
  V - Set if the original operand was 10000000.

C - Set if a borrow is generated; cleared otherwise.
Description: Replaces the operand with its twos complement. The C (carry) bit represents a borrow and is set to the inverse of the resulting binary carry. Note that 80_{16} is replaced by itself and only in this case is the V (overflow) bit set. The value 00_{16} is also replaced by itself, and only in this case is the C (carry) bit cleared.
Addressing Modes: Inherent; Extended; Direct.

NOP

No Operation

Source Form: NOP
Operation: Not affected.

Condition Codes: This instruction causes only the program counter to be incremented. No other registers or memory locations are affected.
Addressing Mode: Inherent.

OR

Inclusive OR Memory into Register

Source Forms: ORA P; ORB P
Operation: R - R v M
Condition Codes:
  H - Not affected.
  N - Set if the result is negative; cleared otherwise.

Z - Set if the result is zero; cleared otherwise
V - Always cleared.
C - Not affected.
Description: Performs an inclusive OR operation between the contents of accumulator A or B and the contents of memory location M and the result is stored in accumulator A or B.
Addressing Modes: Immediate; Extended; Direct; Indexed.

OR

Inclusive OR Memory Immediate into Condition Code Register

Source Form: ORCC #XX
Operation: R' - R v M
Condition Codes: Affected according to the operation.

Description: Performs an inclusive OR operation between the contents of the condition code registers and the immediate value, and the result is placed in the condition code register. This instruction may be used to set interrupt masks (disable interrupts) or any other bit(s).
Addressing Mode: Immediate.

PSHS

Push Registers on the Hardware Stack

Source Form:
  PSHS register list
  PSHS #LABEL
Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

push order →

Operation:
  IFF b7 of postbyte set, then: SP - SP - 1, (SP) - PCL
                                SP' - SP - 1, (SP) - PCH
  IFF b6 of postbyte set, then: SP' - SP - 1, (SP) - USL
                                SP' - SP - 1, (SP) - USH

IFF b5 of postbyte set, then: SP' - SP - 1, (SP) - IYL
                              SP' - SP - 1, (SP) - IYH
IFF b4 of postbyte set, then: SP' - SP - 1, (SP) - IXL
                              SP' - SP - 1, (SP) - IXH
IFF b3 of postbyte set, then: SP' - SP - 1, (SP) - DPR
IFF b2 of postbyte set, then: SP' - SP - 1, (SP) - ACCB
IFF b1 of postbyte set, then: SP' - SP - 1, (SP) - ACCA
IFF b0 of postbyte set, then: SP' - SP - 1, (SP) - CCR
Condition Codes: Not affected.
Description: All, some, or none of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself).
Addressing Mode: Immediate.
Comments: A single register may be placed on the stack with the condition codes set by doing an autodecrement store onto the stack (example: STX,   S)

PSHU

Push Registers on the User Stack

Source Form:
  PSHU register list
  PSHU #LABEL
Postbyte

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

push order →

Operation:
  IFF b7 of postbyte set, then: US' - US - 1, (US) - PCL
                                US' - US - 1, (US) - PCH
  IFF b6 of postbyte set, then: US' - US - 1, (US) - SPL
                                US' - US - 1, (US) - SPH

IFF b5 of postbyte set, then: US' - US - 1, (US) - IYL
                              US' - US - 1, (US) - IYH
IFF b4 of postbyte set, then: US' - US - 1, (US) - IXL
                              US' - US - 1, (US) - IXH
IFF b3 of postbyte set, then: US' - US - 1, (US) - DPR
IFF b2 of postbyte set, then: US' - US - 1, (US) - ACCB
IFF b1 of postbyte set, then: US' - US - 1, (US) - ACCA
IFF b0 of postbyte set, then: US' - US - 1, (US) - CCR
Condition Codes: Not affected.
Description: All, some, or none of the processor registers are pushed onto the user stack (with the exception of the user stack pointer itself).
Addressing Mode: Immediate.
Comments: A single register may be placed on the stack with the condition codes set by doing an autodecrement store onto the stack (example, STX,   U)

Pull Registers from the Hardware Stack

Source Form:
PULS register list
PULS #LABEL
Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

← pull order

Operation:
IFF b0 of postbyte set, then: CCR' ← (SP), SP'← SP + 1
IFF b1 of postbyte set, then: ACCA'← (SP), SP'← SP + 1
IFF b2 of postbyte set, then: ACCB'← (SP), SP'← SP + 1
IFF b3 of postbyte set, then: DPR' ← (SP), SP'← SP + 1
IFF b4 of postbyte set, then: IXH' ← (SP), SP'← SP + 1
IXL' ← (SP), SP'← SP + 1

IFF b5 of postbyte set, then: IYH' ← (SP), SP'← SP + 1
IYL' ← (SP), SP'← SP + 1
IFF b6 of postbyte set, then: USH' ← (SP), SP'← SP + 1
USL' ← (SP), SP'← SP + 1
IFF b7 of postbyte set, then: PCH' ← (SP), SP'← SP + 1
PCL' ← (SP), SP'← SP + 1

Condition Codes: May be pulled from stack; not affected otherwise.
Description: All, some, or none of the processor registers are pulled from the hardware stack (with the exception of the hardware stack pointer itself).
Addressing Mode: Immediate.
Comments: A single register may be pulled from the stack with condition codes set by doing an autoincrement load from the stack (example: LDX,S++).

Pull Registers from the User Stack

Source Form:
PULU register list
PULU #LABEL
Postbyte:

b7	b6	b5	b4	b3	b2	b1	b0
PC	U	Y	X	DP	B	A	CC

← pull order

Operation:
IFF b0 of postbyte set, then: CCR' ← (US), US'← US + 1
IFF b1 of postbyte set, then: ACCA'← (US), US'← US + 1
IFF b2 of postbyte set, then: ACCB'← (US), US'← US + 1
IFF b3 of postbyte set, then: DPR' ← (US), US'← US + 1
IFF b4 of postbyte set, then: IXH' ← (US), US'← US + 1
IXL' ← (US), US'← US + 1

IFF b5 of postbyte set, then: IYH' ← (US), US'← US + 1
IYL' ← (US), US'← US + 1
IFF b6 of postbyte set, then: SPH' ← (US), US'← US + 1
SPL' ← (US), US'← US + 1
IFF b7 of postbyte set, then: PCH' ← (US), US'← US + 1
PCL' ← (US), US'← US + 1

Condition Codes: May be pulled from stack; not affected otherwise.
Description: All, some, or none of the processor registers are pulled from the user stack (with the exception of the user stack pointer itself).
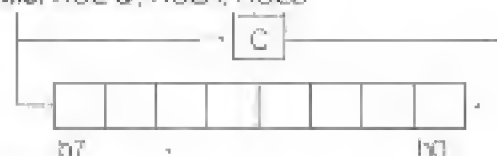**Addressing Mode:** Immediate.
Comments: A single register may be pulled from the stack with condition codes set by doing an autoincrement load from the stack (example: LDX,U++)

Rotate Left

Source Forms: ROL Q; ROLA; ROLB

Operation:

b7 · b0

Condition Codes:
H — Not affected.

N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Loaded with the result of the exclusive OR of bits six and seven of the original operand.
C — Loaded with bit seven of the original operand.
Description: Rotates all bits of the operand one place left through the C (carry) bit. This is a 9-bit rotation.
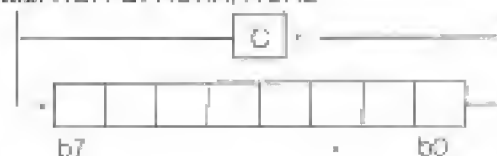**Addressing Mode:** Inherent; Extended; Direct; Indexed.

Rotate Right

Source Forms: ROR Q; RORA; RORB

Operation:

b7 · b0

Condition Codes:
H — Not affected.

N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Not affected.
C — Loaded with bit zero of the previous operand
Description: Rotates all bits of the operand one place right through the C (carry) bit. This is a 9-bit rotation.
Addressing Modes: Inherent; Extended; Direct; Indexed.

RTI — Return from Interrupt

Source Form: RTI

Operation: CCR' ← (SP), SP' ← SP + 1, then
IFF CCR bit E is set, then:
ACCA' ← (SP), SP' ← SP + 1
ACCB' ← (SP), SP' ← SP + 1
DPR' ← (SP), SP' ← SP + 1
IXH' ← (SP), SP' ← SP + 1
IXL' ← (SP), SP' ← SP + 1
IYH' ← (SP), SP' ← SP + 1
IYL' ← (SP), SP' ← SP + 1
USH' ← (SP), SP' ← SP + 1
USL' ← (SP), SP' ← SP + 1

PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1
IFF CCR bit E is clear, then PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

Condition Codes: Recovered from the stack.

Description: The saved machine state is recovered from the hardware stack and control is returned to the interrupted program. If the recovered E (entire) bit is clear, it indicates that only a subset of the machine state was saved (return address and condition codes) and only that subset is recovered.

Addressing Mode: Inherent

RTS — Return from Subroutine

Source Form: RTS

Operation:
PCH' ← (SP), SP' ← SP + 1
PCL' ← (SP), SP' ← SP + 1

Condition Codes: Not affected.

Description: Program control is returned from the subroutine to the calling program. The return address is pulled from the stack.

Addressing Mode: Inherent

SBC — Subtract with Borrow

Source Forms: SBCA P; SBCB P

Operation: R ← R − M − C

Condition Codes:
H — Undefined.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Set if an overflow is generated; cleared otherwise.
C — Set if a borrow is generated; cleared otherwise.

Description: Subtracts the contents of memory location M and the borrow (in the C (carry) bit) from the contents of the designated 8-bit register, and places the result in that register. The C bit represents a borrow and is set to the inverse of the resulting binary carry.

Addressing Modes: Immediate; Extended; Direct; Indexed

SEX — Sign Extended

Source Form: SEX

Operation:
If bit seven of ACCB is set then ACCA' ← FF_{16}
else ACCA' ← 00_{16}

Condition Codes:
H — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Not affected.
C — Not affected.

Description: This instruction transforms a twos complement 8-bit value in accumulator B into a twos complement 16-bit value in the D accumulator.

Addressing Mode: Inherent

ST (8-Bit) — Store Register into Memory

Source Forms: STA P; STB P

Operation: M ← R

Condition Codes:
H — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Always cleared.
C — Not affected.

Description: Writes the contents of an 8-bit register into a memory location.

Addressing Modes: Extended; Direct; Indexed.

ST (16-Bit) — Store Register into Memory

Source Forms: STD P; STX P; STY P; STS P; STU P

Operation: M:M + 1 ← R

Condition Codes:
H — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Always cleared.
C — Not affected.

Description: Writes the contents of a 16-bit register into two consecutive memory locations.

Addressing Modes: Extended; Direct; Indexed.

SUB (8-Bit) — Subtract Memory from Register

Source Forms: SUBA P; SUBB P

Operation: R ← R − M

Condition Codes:
H — Undefined.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Set if the overflow is generated; cleared otherwise.
C — Set if a borrow is generated; cleared otherwise.

Description: Subtracts the value in memory location M from the contents of a designated 8-bit register. The C (carry) bit represents a borrow and is set to the inverse of the resulting binary carry.

Addressing Modes: Immediate; Extended; Direct; Indexed.

Subtract Memory from Register

Source Forms: SUBD P
Operation: R' ← R − M:M + 1
Condition Codes:
H — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero, cleared otherwise.
V — Set if the overflow is generated; cleared otherwise.
C — Set if a borrow is generated; cleared otherwise.
Description: Subtracts the value in memory location M:M + 1 from the contents of a designated 16-bit register. The C (carry) bit represents a borrow and is set to the inverse of the resulting binary carry.
Addressing Modes: Immediate, Extended; Direct; Indexed.

SUB (16-Bit)

Software Interrupt

Source Form: SWI
Operation:
Set E (entire state will be saved)
SP' ← SP − 1, (SP) ← PCL
SP' ← SP − 1, (SP) ← PCH
SP' ← SP − 1, (SP) ← USL
SP' ← SP − 1, (SP) ← USH
SP' ← SP − 1, (SP) ← IYL
SP' ← SP − 1, (SP) ← IYH
SP' ← SP − 1, (SP) ← IXL
SP' ← SP − 1, (SP) ← IXH
SP' ← SP − 1, (SP) ← DPR
SP' ← SP − 1, (SP) ← ACCB
SP' ← SP − 1, (SP) ← ACCA
SP' ← SP − 1, (SP) ← CCR
Set I, F (mask interrupts)
PC' ← (FFFA):(FFFB)
Condition Codes: Not affected.
Description: All of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself), and control is transferred through the software interrupt vector. Both the normal and fast interrupts are masked (disabled).
Addressing Mode: Inherent.

SWI

Software Interrupt 2

Source Form: SWI2
Operation:
Set E (entire state saved)
SP' ← SP − 1, (SP) ← PCL
SP' ← SP − 1, (SP) ← PCH
SP' ← SP − 1, (SP) ← USL
SP' ← SP − 1, (SP) ← USH
SP' ← SP − 1, (SP) ← IYL
SP' ← SP − 1, (SP) ← IYH
SP' ← SP − 1, (SP) ← IXL
SP' ← SP − 1, (SP) ← IXH
SP' ← SP − 1, (SP) ← DPR
SP' ← SP − 1, (SP) ← ACCB
SP' ← SP − 1, (SP) ← ACCA
SP' ← SP − 1, (SP) ← CCR
PC' ← (FFF4):(FFF5)
Condition Codes: Not affected.
Description: All of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself), and control is transferred through the software interrupt 2 vector. This interrupt is available to the end user and must not be used in packaged software. This interrupt does not mask (disable) the normal and fast interrupts.
Addressing Mode: Inherent

SWI2

Software Interrupt 3

Source Form: SWI3
Operation:
Set E (entire state will be saved)
SP' ← SP − 1, (SP) ← PCL
SP' ← SP − 1, (SP) ← PCH
SP' ← SP − 1, (SP) ← USL
SP' ← SP − 1, (SP) ← USH
SP' ← SP − 1, (SP) ← IYL
SP' ← SP − 1, (SP) ← IYH
SP' ← SP − 1, (SP) ← IXL
SP' ← SP − 1, (SP) ← IXH
SP' ← SP − 1, (SP) ← DPR
SP' ← SP − 1, (SP) ← ACCB
SP' ← SP − 1, (SP) ← ACCA
SP' ← SP − 1, (SP) ← CCR
PC' ← (FFF2):(FFF3)
Condition Codes: Not affected.
Description: All of the processor registers are pushed onto the hardware stack (with the exception of the hardware stack pointer itself), and control is transferred through the software interrupt 3 vector. This interrupt does not mask (disable) the normal and fast interrupts.
Addressing Mode: Inherent.

SWI3

SYNC

Synchronize to External Event

Source Form: SYNC
Operation: Stop processing instructions
Condition Codes: Not affected
Description: When a SYNC instruction is executed, the processor enters a synchronizing state, stops processing instructions, and waits for an interrupt. When an interrupt occurs, the synchronizing state is cleared and processing continues. If the interrupt is enabled, and it last three cycles or more, the processor will perform the interrupt routine. If the interrupt is masked or is shorter than three cycles, the processor simply continues to the next instruction. While in the synchronizing state, the address and data buses are in the high-impedance state.

This instruction provides software synchronization with a hardware process. Consider the following example for high-speed acquisition of data:

```
FAST    SYNC              WAIT FOR DATA
        Interrupt!
        LDA    DISC       DATA FROM DISC AND
                          CLEAR INTERRUPT
        STA    ,X+        PUT IN BUFFER
        DECB              COUNT IT, DONE?
        BNE    FAST       GO AGAIN IF NOT.
```

The synchronizing state is cleared by any interrupt. Of course, enabled interrupts at this point may destroy the data transfer and, as such, should represent only emergency conditions.

The same connection used for interrupt-driven I/O service may also be used for high-speed data transfers by setting the interrupt mask and using the SYNC instruction as the above example demonstrates.
Addressing Mode: Inherent

TFR

Transfer Register to Register

Source Form: TFR R1, R2
Operation: R1 → R2
Condition Code: Not affected unless R2 is the condition code register.
Description: Transfers data between two designated registers. Bits 7-4 of the postbyte define the source register, while bits 3-0 define the destination register, as follows:

0000 – A:B		1000 – A	
0001 – X		1001 – B	
0010 – Y		1010 – CCR	
0011 – US		1011 – DPR	
0100 – SP		1100 – Undefined	
0101 – PC		1101 – Undefined	
0110 – Undefined		1110 – Undefined	
0111 – Undefined		1111 – Undefined	

Only like size registers may be transferred (8-bit to 8-bit, or 16-bit to 16-bit.)
Addressing Mode: Immediate

TST

Test

Source Forms: TST Q; TSTA, TSTB
Operation: TEMP ← M – 0
Condition Codes:
H — Not affected
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Always cleared
C — Not affected

Description: Set the N (negative) and Z (zero) bits according to the contents of memory location M, and clear the V (overflow) bit. The TST instruction provides only minimum information when testing unsigned values; since no unsigned value is less than zero, BLO and BLS have no utility. While BHI could be used after TST, it provides exactly the same control as BNE, which is preferred. The signed branches are available.

Addressing Modes: Inherent; Extended; Direct; Indexed.
Comments: The MC6800 processor clears the C (carry) bit.

FIRQ

Fast Interrupt Request (Hardware Interrupt)

Operation:
```
IFF F bit clear, then: SP' ← SP – 1, (SP) ← PCL
                       SP' ← SP – 1, (SP) ← PCH
                       Clear E (subset state is saved)
                       SP' ← SP – 1, (SP) ← CCR
                       Set F, I (mask further interrupts)
                       PC' ← (FFF6):(FFF7)
```
Condition Codes: Not affected
Description: A FIRQ (fast interrupt request) with the F (fast interrupt request mask) bit clear causes this interrupt sequence to occur at the end of the current instruction. The program counter and condition code register are pushed onto the hardware stack. Program control is transferred through the fast interrupt request vector. An RTI (return from interrupt) instruction returns the processor to the original task. It is possible to enter the fast interrupt request routine with the entire machine state saved if the fast interrupt request occurs after a clear and wait for interrupt instruction. A normal interrupt request has lower priority than the fast interrupt request and is prevented from interrupting the fast interrupt request routine by automatic setting of the I (interrupt request mask) bit. This mask bit could then be reset during the interrupt routine if priority was not desired. The fast interrupt request allows operations on memory. TST, INC, DEC, etc. instructions without the overhead of saving the entire machine state on the stack.
Addressing Mode: Inherent

Interrupt Request
(Hardware Interrupt)

Operation:

IFF I bit clear, then: SP' - SP - 1, (SP)- PCL

SP'- SP - 1, (SP)- PCH

SP'- SP - 1, (SP)- USL

SP'- SP - 1, (SP)- USH

SP'- SP - 1, (SP)- IYL

SP'- SP - 1, (SP)- IYH

SP'- SP - 1, (SP)- IXL

SP'- SP - 1, (SP)- IXH

SP'- SP - 1, (SP)- DPR

SP'- SP - 1, (SP)- ACCB

SP'- SP - 1, (SP)- ACCA

Set E (entire state saved)

SP'- SP - 1, (SP)- CCR

Set I (mask further IRQ interrupts)

PC'- (FFF8):(FFF9)

Condition Codes: Not affected.

Description: If the I (interrupt request mask) bit is clear, a low level on the IRQ input causes this interrupt sequence to occur at the end of the current instruction. Control is returned to the interrupted program using a RTI (return from interrupt) instruction. A FIRQ (fast interrupt request) may interrupt a normal IRQ (interrupt request) routine and be recognized anytime after the interrupt vector is taken.

Addressing Mode: Inherent.

Non-Maskable Interrupt
(Hardware Interrupt)

Operation:

SP'- SP - 1, (SP)- PCL

SP'- SP - 1, (SP)- PCH

SP'- SP - 1, (SP)- USL

SP'- SP - 1, (SP)- USH

SP'- SP - 1, (SP)- IYL

SP'- SP - 1, (SP)- IYH

SP'- SP - 1, (SP)- IXL

SP'- SP - 1, (SP)- IXH

SP'- SP - 1, (SP)- DPR

SP'- SP - 1, (SP)- ACCB

SP'- SP - 1, (SP)- ACCA

Set E (entire state save)

SP'- SP - 1, (SP)- CCR

Set I, F (mask interrupts)

PC -- (FFFC):(FFFD)

Condition Codes: Not affected.

Description: A negative edge on the NMI (non-maskable interrupt) input causes all of the processor's registers (except the hardware stack pointer) to be pushed onto the hardware stack, starting at the end of the current instruction. Program control is transferred through the NMI vector. Successive negative edges on the NMI input will cause successive NMI operations. Non-maskable interrupt operation can be internally blocked by a RESET operation and any non-maskable interrupt that occurs will be latched. If this happens, the non-maskable interrupt operation will occur after the first load into the stack pointer (LDS; TFR r,s; EXG r,s; etc.) after RESET.

Addressing Mode: Inherent.

Restart (Hardware Interrupt)

Operation:

CCR - X1X1XXXX

DPR'- 00₁₆

PC'- (FFFE):(FFFF)

Condition Codes: Not affected.

Description: The processor is initialized (required after power-on) to start program execution. The starting address is fetched from the restart vector.

Addressing Mode: Extended; Indirect.

Appendix A / Editor Commands

Definition of Terms

line
A line number in the program. Any lines between 0-63999 may be used. These symbols may be used:

#	First line in the program.
*	Last line in the program.
.	Current line (see definition below).

current line
The last line inserted, edited, or printed.

startline
The line where an operation will begin. In most commands *startline* is optional. If omitted, the *current line* is used.

range
The line or lines to use in an operation. If more than one line are in the range, they must be specified with one of these symbols:

:	to separate the startline from the ending line
!	to separate the startline from the number of lines

increment
The increment to use between lines. In most commands, *increment* is optional. If omitted, the last specified *increment* is used. On start-up, *increment* is set to 10.

filename
A 1-8 character name of a tape file.

COMMANDS	PAGES DISCUSSED
C*startline,range,increment* Copies *range* to a new location beginning with *startline* using the specified *increment*. *startline*, *range*, and *increment* must all be included. `C500,100:150,10`	11
D*range* Deletes *range*. If *range* is omitted, current line is deleted. `D100    D100:150    D`	11
E*line* Enters a line for editing. If *line* is omitted, current line is used. `E100    E`	10

These are the editing subcommands:

A	Cancels all changes and restarts the edit.
nC**string**	Changes *n* characters to *string*. If *n* is omitted, changes the character at the current cursor position.
nD	Deletes *n* characters. If *n* is omitted, deletes character at current cursor position.
E	Ends line editing and enters all changes without displaying the rest of the line.
H	Deletes rest of line and allows insert.
I *string*	Inserts *string* starting at the current cursor position. While in this Mode, (←) deletes a character.

COMMANDS	PAGES DISCUSSED

nKcharacter   Deletes all characters from the current cursor position to the *n*th occurrence of *character*. If *n* is omitted, deletes to the first occurrence.

L   Lists current line and continues edit.

O   Quits the edit and ignores all changes.

nScharacter   Searches for *n*th occurrence of *character*. If *n* is omitted, searches for first occurrence.

X   Extends line.

(ENTER)   Ends line editing, enters all changes and displays the rest of the line.

(SHIFT) (←)   Escape from subcommand.

n(SPACEBAR)   Moves cursor *n* characters to the right. If *n* is omitted, moves one space.

n(−)   Moves cursor *n* positions to the left. If *n* is omitted, moves the cursor one position.

Fstring
Finds the *string* of characters. Search begins with the current line and ends each time the *string* is found. If *string* is omitted, the last string defined is used.

   FABC   F

Hrange
Prints *range* on the Printer. If *range* is omitted, current line is printed.   10

   H100   H100:200   H

Istartline,increment
Inserts lines beginning at *startline* using the specified *increment*. *startline* and *increment* are optional.   11

   I150,5   I200   I,10

L filename
Loads the specified text file from cassette tape. If *filename* is omitted, the next file is loaded.   10

   L SAMPLE   L

Mstartline,range,increment
Move command, works like copy except the original lines are deleted.

Nstartline,increment
Renumbers beginning at *startline*, using the specified *increment*. *startline* and *increment* are optional.   11

   N100,50   N100   N

Prange
Displays *range* on the screen.   10

   P100:200   P100'5   P=   P*
   P

Q
Returns to BASIC. Type EXEC 49152 to return to Editor from BASIC.   11

Rstartline,increment
Allows you to replace *startline*, and then insert lines using *increment*. *startline* and *increment* are optional.   11

   R100,10   R100   R

COMMANDS	PAGES DISCUSSED

T*range*　　　　　　　　　　　　　　　　　　　　　　　10

Prints *range* on the printer, without including the line numbers.

　　T100　　T100:500

V*filename*

Verifies *filename* to ensure that it is free of checksum errors. Works like BASIC's SKIPF command. If *filename* is omitted, verifies next file found.

　　VTEST

Z　　　　　　　　　　　　　　　　　　　　　　　　　5, 11

Go to ZBUG.

(▲)

Scrolls up in memory.

(▼)

Scrolls down in memory.

Appendix B/Assembler Command & Switches

COMMAND/SWITCH	PAGES DISCUSSED
A *filename switch* ...	13
Assembles the text program into machine code. Any of the following switches may be used:	

/AO	Absolute Origin. (Applies only if /IM is set.)	15
/IM	In Memory Assembly.	13
/LP	Assembly listing on the printer.	13
/MO	Manual Origin. (Applies only if /IM is set.)	15
/NL	No listing printed.	13
/NO	No object code generated.	13
/NS	No symbol table generated.	13
/SS	Short screen.	13
/WE	Wait on assembly errors.	13

Unless the /IM switch is used, the program will be assembled on tape using the specified one to eight character filename. If *filename* is omitted, NONAME is used.

Examples

```
A SAMPLE/IM
A
A/IM/AO
```

Appendix C / ZBUG Commands

Definition of Terms

expression

One or more numbers, symbols, or ASCII characters. If more than one are used, you may separate them with these operators:

Multiplication	*	Addition	+
Division	.DIV.	Subtraction	−
Modulus	.MOD.	Equals	.EQU.
Shift	<	Not Equal	.NEQ.
Local And	.AND.	Positive	+
Exclusive Or	.XOR.	Negative	−
Logical Or	.OR.	Complement	.NOT

address

A location in memory. This may be specified as an expression using either numbers or symbols.

filename

A one to eight character name of a tape file.

COMMANDS	PAGES DISCUSSED

C   18
Continues execution of the program after interruption at a breakpoint.

D   18
Displays all the breakpoints that have been set.

E
Exits ZBUG and enters the Editor.

G*address*   18
Executes the program beginning at *address*.

L*filename* (ENTER)   19
Loads the machine-code file from cassette tape. If *filename* is omitted, the next file is loaded.

P*filename first address last address start execution address*   19
Saves the contents of memory from start address to *ending address* on tape. *execution address* specifies the address where the program being saved begins execution.

R   18
Displays the contents of all the registers.

T*address1 address2*   19
Displays the memory locations from *address1* to *address2*, inclusive.

TH*address1 address2*   19
Prints the memory locations from *address1* to *address2*, inclusive.

U*source address destination address count*
Transfers the contents of memory beginning at *source address* and continuing for count bytes to another location in memory beginning with *destination address*.

V*filename*
Verifies date on the specified file or the next file on the tape if no *filename* is specified.

COMMANDS	PAGES DISCUSSED

Xaddress   18
Sets a breakpoint at *address*. If *address* is omitted, the current location will be used.

Yaddress   18
Deletes the breakpoint at the specified address. If *address* is omitted, all breakpoints are deleted.

Examination Mode Commands

A	ASCII Mode	5
B	Byte Mode	5
M	Mnemonic Mode	6
W	Word Mode	5

(the default is M)

Display Mode Commands

H	Half Symbolic	17
N	Numeric	17
S	Symbolic	17

(the default is S)

Numbering System Mode Commands

*O*base	Output	21
*I*base	Input	21

(base can be 8, 10, or 16. The default is 16.)

Special Symbols

address/   5
register/   18

Opens address or register and displays its contents. If address or register is omitted, the last address opened will be re-opened. After the contents have been displayed, you may type:

New contents	To change the contents.	6
(ENTER)	To close and enter any change.	6
(BREAK)	To close and delete any change.	
(←)	To open next address and enter any change.	5
(→)	To open preceding address.	5
(↓)	To branch to the address pointed to by the instruction beginning at the current location.	
;	To force numeric display mode.	17
=	To force numeric and byte modes.	
:	To force flags.*	

address,   18
Executes *address*. If *address* is omitted, the next instruction is executed.

expression=   21
Calculates *expression* and displays the results

*The colon does not actually have anything to do with the CC (status flag) register. It simply interprets the contents of the given address AS IF it contained flag bits.

Appendix D / Editor Error Messages

The following are descriptions of the error messages you can get while in the Editor, Assembler, or ZBUG:

BAD BREAKPOINT *(ZBUG)*
You are attempting to set a breakpoint (1) greater than 7, (2) in ROM, (3) at a SWI command, (4) at an address where one is already set.

BAD COMMAND *(Editor)*
An illegal command letter was used on the command line.

BAD COMMAND *(ZBUG)*
You are not using a ZBUG command.

BAD LABEL *(Assembler)*
The symbol you are using is (1) not a legal symbol, (2) not terminated with either a space, a tab, or a carriage return, or (3) has been used with ORG or END, which do not allow labels. (4) longer than six characters.

BAD LINE NUMBER *(Editor)*
You are using a line number that is not in the range of 1-63999. If you are loading a file from tape, this could mean the tape is bad or the tape does not contain a TEXT file.

BAD MEMORY *(Assembler)*
You are attempting to do an in-memory assembly which would (1) overwrite system memory (an address lower than hexadecimal 0600), (2) overwrite the edit buffer or symbol table, (3) go into the protected area set by USRORG, or (4) go over the top of RAM.

If using the /AO switch, check to see that you've included an ORG instruction. When using /MO, check the addresses you set for BEGTEMP and USRORG. This could also be caused by the data not being stored correctly because of some code generated by an in-memory assembly. See the Chapter on Assembling for more information.

BAD MEMORY *(ZBUG)*
The data did not store correctly on a memory modification. This error will occur if you try to modify ROM addresses, or store anything beyond MAXMEM.

BAD OPCODE *(Assembler)*
The op code is either not valid or is not terminated with a space, a tab or a carriage return.

BAD OPERAND *(Assembler)*
There is some syntax error in the operand field. See the syntax for the instruction in Section II.

BAD PARAMETERS *(Editor)*
Usually, this means your command line has a syntax error.

BAD PARAMETERS *(ZBUG)*
You have specified a filename greater than eight characters.

BAD RADIX *(ZBUG)*
You have specified a numbering system other than 10, 8 or 16.

BUFFER FULL *(Editor)*
There is not enough room in the Edit Buffer for another line of text.

BUFFER EMPTY *(Editor)*
The specified command requires that there be some text in the Edit Buffer, and there isn't any.

BYTE OVERFLOW *(Assembler)*
There is a field overflow in an 8-bit data quantity in an immediate operand, an offset, a short branch, or an FCB pseudo op.

DP ERROR *(Assembler)*
Direct Page error. The high order byte of an operand where direct addressing has been forced (<) does not match the value set by the most recent SETDP pseudo op.

EXPRESSION ERROR *(Assembler and ZBUG)*
Same kind of syntax error in an expression or division by zero.

FM ERROR *(Editor and ZBUG)*
File Mode Error. The file you are attempting to load is not a TEXT file (if in the Editor) or a CODE file (if in ZBUG).

I/O ERROR *(Editor and ZBUG)*
Input/Output error. A checksum error was encountered while loading a file from a cassette tape. The tape may be bad, or the volume setting may be wrong. Try higher.

MISSING END *(Assembler)*
Every assembly language must have END as its last command.

MISSING INFORMATION *(Assembler)*
(1) There is a missing delimiter in an FCC pseudo op, or (2) There is no label on a SET or EQU pseudo op.

MISSING OPERAND *(Assembler)*
One or more operands are missing from a command requiring one.

MULTIPLY DEFINED SYMBOL *(Assembler)*
A label has been defined more than one time.

NO ROOM BETWEEN LINES *(Editor)*
There is not enough room between lines to use the increment you've specified. Specify a smaller increment or renumber (N) the text using a larger increment. Remember that the last increment you used is kept until you specify a new one.

NO SUCH LINES *(Editor)*
The specified line or lines do not exist.

REGISTER ERROR *(Assembler)*
(1) No registers have been specified with a PSH/PUL instruction, (2) A register has been specified more than once in a PSH/PUL instruction, or (3) There is a register mis-match with an EXG/TFR instruction.

SEARCH FAILS *(Editor)*
The string specified in the Find (F) command could not be found in the edit buffer, beginning with the line speci-

fied. If no line is specified the current line will be used.

SYMBOL TABLE OVERFLOW *(Assembler)*
(1) The symbol table will extend past USRORG into the protected area of memory. (2) There is not enough room between BEGTMP and USRORG for the edit buffer and symbol table. At least 300 hexadecimal bytes must be allowed for BEGTMP. (See the chapter on Assembling.)

UNDEFINED SYMBOL *(Assembler)*
The symbol in the program was never listed in the label field or defined with an EQU statement.

Appendix E / Memory Map

DECIMAL	HEX 14B	CONTENTS	DESCRIPTION
0-105	0-69	Direct Page RAM	Can be used for machine-code programs.
112-255	70-FF		Cannot be used for machine-code programs.
256-273	100-111	Internal Use	Interrupt vectors.
274-276	112-114	USRJMP	Jump to BASIC's USR routine.
277-281	115-119		Can be used for machine-code programs.
282	11A	Keyboard Alpha Lock	0 - not locked; FF - locked.
283-284	11B-11C	Keyboard Delay Constant	
285-337	11D-151		Can be used by machine-code programs.
338-345	152-159	Keyboard Rollover Tables	
346-349	15A-15D	Joystick Pot Values	
350-1023	15E-3FF	Internal Use	
1024-1535	0400-05FF	Video Text Memory	
1536-top of RAM top of RAM is 16383 for 16K systems; 32767 for 32K systems	0600-top of RAM top of RAM is 3FFF for 16K systems; 7FFF for 32K systems	If the Editor-Assembler is in control, it allocates these Random Access memory addresses in this manner (see the /MO and /AO switch in *Chapter 4* for information on how to change this):	
		1. Temporaries	Space reserved for temporary storage of EDTASM's variables buffers, and stacks (this consumes hexadecimal 200 bytes).
		2. Edit Buffer	Storage space for the program lines you insert with the Editor.
		3. Symbol Table	Storage space for all the symbols in your program and their corresponding values.
		4. Object Code	Storage space for your assembled program.
		If BASIC is in control, it allocates these Random Access memory locations in this manner:	
		1. Graphics Video Memory	Space reserved for graphics video pages. 6144 bytes or 4 pages are reserved for this on start-up. This value can be reset by the PCLEAR statement; number of pages reserved by PCLEAR X 1,536 bytes per page. (Note: All pages must start at a 256-byte page boundary — i.e., a memory location divisible by 256.)
		2. BASIC Program Storage 3. BASIC Variable Storage 4. Stack	Space reserved for BASIC Programs and Variables. 6455* bytes (16K systems) or 22,839* bytes (32K systems) are reserved for this on start-up. This value can be reset by different settings of Random File Buffers, FCBs, Graphics Video Memory, String Space or User Memory.
		5. String Space	Total space for string data. On start-up, 200 bytes are reserved, but this can be reset by the CLEAR statement.
		6. User Memory	Total space for user machine-language routines. No space is reserved for this on start-up, but this can be reset by the CLEAR statement.
32768-40959	8000-9FFF	Extended COLOR BASIC ROM	Read Only Memory
40960-49151	A000-BFFF	COLOR BASIC ROM	Read Only Memory
49152-57343	C000-DFFF	EDTASM+ ROM	Read Only Memory
57344-65279	E000-FEFF	Unused	
65280-65535	FF00-FFFF	Input/Output	

Appendix F / ROM Routines

The Color BASIC ROM contains many subroutines that can be called by a machine-language program. Each subroutine will be described in the following format:

NAME — *Entry address*
Operation Performed
Entry Condition
Exit Condition

> *Note: The subroutine **NAME** is only for reference. It is not recognized by the Color Computer. The **entry address** is given in hexadecimal form; you must use an indirect jump to this address. **Entry** and **Exit Conditions** are given for machine-language programs.*

BLKIN = [A006]
Reads a Block from Cassette

Entry Conditions
Cassette must be on and in bit sync (see CSRDON). CBUFAD contains the buffer address.

Exit Conditions
BLKTYP, which is located at 7C, contains the block type:
    0 = File Header
    1 = Data
    FF = End of File
BLKLEN, located at 7D, contains the number of data bytes in the block (0-255).
Z* = 1, A = CSRERR = 0 (if no errors).
Z = 0, A = CSRERR = 1 (if a checksum error occurs).
Z = 0, A = CSRERR = 2 (if a memory error occurs).

> *Note: CSRERR = 81*

Unless a memory error occurs, X = CBUFAD + BLKLEN. If a memory error occurs, X points to beyond the bad address. Interrupts are masked. U and Y are preserved, all other modified.

Z is a flag in the Condition Code (CC) register.

BLKOUT = [A008]
Writes a Block to Cassette

Entry Conditions
The tape should be up to speed and a leader of hex 55s should have been written if this is the first block to be written after a motor-on.
CBUFAD, located at 7E, contains the buffer address.
BLKTYP, located at 7C, contains the block type.
BLKLEN, located at 7D, contains the number of data bytes.

Exit Conditions
Interrupts are masked.
X = CBUFAD + BLKLEN.
All registers are modified.

WRTLDR = [A00C]
Turns the Cassette On and Writes a Leader

Entry Conditions
None

Exit Conditions
None

CHROUT = [A002]
Outputs a Character to Device

CHROUT outputs a character to the device specified by the contents of 6F (DEVNUM).
DEVNUM = - 2 (printer)
DEVNUM = 0 (screen)

Entry Conditions
On entry, the character to be output is in A.

Exit Conditions
All registers except CC are preserved.

CSRDON = [A004]
Starts Cassette

CSRDON starts the cassette and gets into bit sync for reading.

Entry Conditions
None

Exit Conditions
FIRQ and IRQ are masked. U and Y are preserved. All others are modified.

GIVABF = [B4F4]
Passes parameter to BASIC

Entry Conditions
D = parameter

Exit Conditions
USR variable = parameter

INTCNV = [B3ED]
Passes parameter from BASIC

Entry Conditions
USR argument = parameter

Exit Conditions
D = parameter

JOYIN = [A00A]
Samples Joystick Pots

JOYIN samples all four joystick pots and stores their values in POTVAL through POTVAL + 3.

	Left Joystick	
Up/Down		15A
Right/Left		15B
	Right Joystick	
Up/Down		15C
Right/Left		15D

For Up/Down, the minimum value = UP.
For Right/Left, the minimum value = LEFT.

Entry Conditions
None

Exit Conditions
Y is preserved. All others are modified.

POLCAT = [A000]
Polls Keyboard for a Character

Entry Conditions
None

Exit Conditions
Z = 1, A = 0 (if no key seen).
Z = 0, A = key code, (if key is seen).
B and X are preserved. All others are modified.

INDEX